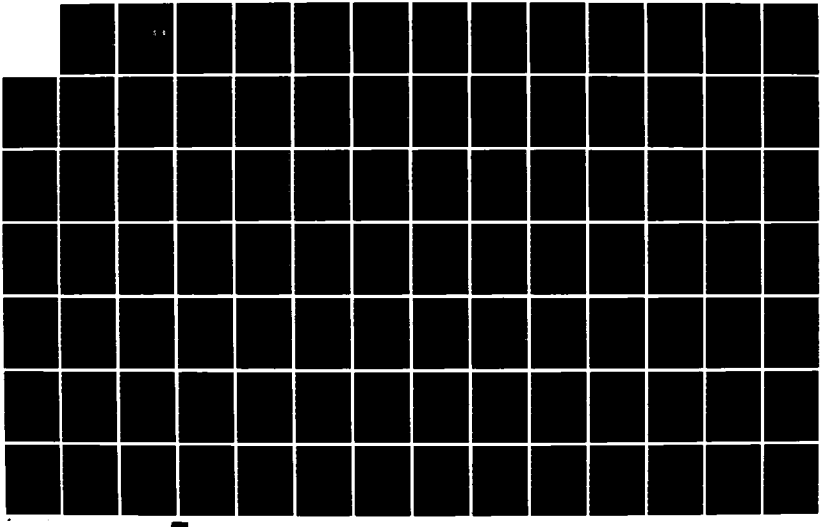
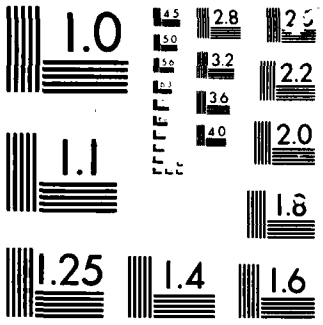
AD-A166 936 MESSAGE PASSING ON A LOCAL NETWORK(U) STANFORD UNIV CA 1/2
DEPT OF COMPUTER SCIENCE W ZWAENEPOEL OCT 85
STAN-CS-85-1083 MDA903-80-C-0102

UNCLASSIFIED F/G 17/2 NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY
CHART

October 1985

Report No. STAN-CS-85-1083
Also numbered CSL-85-283

(2)

Message Passing on a Local Network

by

Willy Zwaenepoel

Department of Computer Science

Stanford University
Stanford, CA 94305

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Message Passing on a Local Network		5. TYPE OF REPORT & PERIOD COVERED
		6. PERFORMING ORG. REPORT NUMBER STAN-CS-85-1083
7. AUTHOR(s) Willy Zwaenepoel		8. CONTRACT OR GRANT NUMBER(s) N00039-83-K-0431
9. PERFORMING ORGANIZATION NAME AND ADDRESS Departments of Computer Science and Electrical Eng. Stanford University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency		12. REPORT DATE
		13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Navalex		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Unlimited

18. SUPPLEMENTARY NOTES

19 KEY WORDS (Continue on reverse side if necessary and identify by block number)

20 ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis focuses on understanding the cost of transparent interprocess communication in a distributed system consisting of a set of machines connected by a local network. Interprocess communication is transparent if processes can communicate without regard to physical host boundaries. Transparent interprocess communication is a very powerful tool because it allows us to view the collection of different machines as a single, logically unified computer system. We concentrate on the efficiency aspects of transparent interprocess communication on a local network.

DD FORM 1 JAN 73 1473

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

In order to obtain experimental evidence, a transparent message-passing mechanism has been implemented as part of the distributed V kernel. This message-passing mechanism has been used as the basis for various distributed applications. In particular, it has been used extensively for providing transparent file access from diskless workstations to a set of network-based file servers. Based in part on experience gained from the implementation and use of the distributed V kerned, this thesis presents four contributions:

1. An empirical evaluation of high-performance message passing on a local network.

2. A queueing network model of file access from diskless workstations over a local area network to a set of file servers.

3. An analysis of the protocol used to support the V interprocess communication on a broadcast network.

4. The integration of the broadcast and multicast capabilities of local area networks into message-based interprocess communication.

Message Passing on a Local Network

Willy Zwaenepoel

Abstract

This thesis focuses on understanding *the cost of transparent interprocess communication* in a distributed system consisting of a set of machines connected by a local network. Interprocess communication is *transparent* if processes can communicate without regard to physical host boundaries. Transparent interprocess communication is a very powerful tool because it allows us to view the collection of different machines as a single, logically unified computer system. We concentrate on the *efficiency* aspects of transparent interprocess communication on a local network.

In order to obtain experimental evidence, a transparent message-passing mechanism has been implemented as part of the *distributed V kernel*. This message-passing mechanism has been used as the basis for various distributed applications. In particular, it has been used extensively for providing transparent file access from diskless workstations to a set of network-based file servers. Based in part on experience gained from the implementation and use of the distributed V kernel, this thesis presents four contributions:

1. An empirical evaluation of high-performance message passing on a local network.
2. A queueing network model of file access from diskless workstations over a local area network to a set of file servers.
3. An analysis of the protocol used to support the V interprocess communication on a broadcast network.
4. The integration of the broadcast and multicast capabilities of local area networks into message-based interprocess communication.

Acknowledgements

I wish to thank my advisor David Cheriton for many things, especially for suggesting some of the topics in this thesis as potential research areas and for providing an experimental environment in which the value of these ideas could be tested. I also wish to thank the other members of my thesis committee -- Forest Baskett, Keith Lantz and Ingolf Lindau -- whose suggestions added both to the contents as well as improved the presentation of this thesis. Some of the material in this thesis grew out of work with a number of different people on joint papers. The contributions of my co-authors -- David Cheriton, Keith Lantz, Ed Lazowska and John Zahorjan -- are gratefully acknowledged.

This thesis could not be what it has become without the contribution of the members of the Distributed Systems Research Group at Stanford. They all deserve thanks in some way or another, in particular Eric Berglund, Steve Deering, Tim Mann, Bill Nowicki, Paul Roy and Marvin Theimer. Many other people provided advice and encouragement in both technical and non-technical ways. They include Thomas Blank, Thomas Gross, Mark Horowitz, Amy Lansky, Ed Lazowska, Keith Marzullo and Jerry Popek. Finally, the contribution of my parents to my education shall not go unnoticed.

Financial support for my work at Stanford was provided by a graduate fellowship from the Belgian American Educational Foundation, by a teaching fellowship from the Stanford Computer Science Department, and by a research assistantship funded by IBM under the Joint Study on Distributed Computing under contracts SEL 47-79, 2-80, 8-81, 18-81 and DARPA under grants MDA 903-80-C-0102 and N 00039-83-K-0431.

Table of Contents

List of Figures

List of Tables

— 1 —
Introduction

1.1. Introduction to the Research Area

In recent years, many researchers have come to the belief that a large, single-machine *timesharing* system is no longer the most effective way of providing computing services for a large class of users. Two technological advances have stimulated this evolution:

1. Advances in VLSI technology have significantly reduced the cost of processors and memory, to the point where it has become economically feasible to dedicate a whole processor with a large amount of memory to a single user.

2. High-speed local networks have made it possible to interconnect such personal machines to expensive peripherals such as printers and file servers, so that the high cost of these peripherals can be amortized over a large number of users.

As a result of these technological advances, *personal computer* based systems have become popular: each user is given a dedicated personal computer which is connected by a local network to specialized server machines like file servers, printers, high-speed computing engines, etc. These new systems have relieved several of the problems inherent to large single-machine timesharing systems:

1. Performance is less dependent on the number of users of the system. Some contention remains for access to the shared server machines, but its effects are typically less pronounced than the performance degradation experienced when a comparable workload is presented to a single-machine timesharing system.

2. The system is easily upgradable in terms of computing power by simply adding one or more extra processors.

3. The dedication of a whole machine to a single user has provided enough spare cycles to supply the user with a high-quality user interface.

However, these systems require explicit action to be taken when access to a remote machine is desired. For example, a file on a remote file server cannot be accessed by the same procedure as the one used to access a local file. Instead, a *file transfer* program has to be invoked to transfer the file from the file server to the local machine, before normal file operations can be performed. In general, access to resources is not *transparent*. Resources on other machines (such as files or virtual terminals) can only be accessed by different mechanisms than those used for accessing similar resources on the local machine.

Separating users onto different machines, with no transparent way to communicate between them, has significantly reduced the amount of *sharing* that can be accomplished among users, both in the sense of physical sharing of resources as well as in the psychological sense of sharing a single environment. In a well-designed timesharing system, the degree of physical sharing is nearly complete: every processor cycle, every byte of memory, every sector of the disk can in principle be used by any user. In a personal computer environment, a user is essentially limited to the resources of his own workstation. Access to resources on other machines is either too awkward or too costly to be worthwhile. Additionally, timesharing systems provide a strong sense of community among their users, particularly because of the ease with which files can be shared between different users.

Lack of transparency, and the resulting loss of sharing, have made it difficult to provide personal computer users with a number of services that are commonplace in good timesharing systems. Some specific examples:

1. In a timesharing system, the executable images of system programs are stored on disk exactly once, and this single copy is shared by all users. If, as is common in the personal computer model, programs can only be loaded from the local disk, then system programs need to be replicated on all workstations. This results in a serious loss of (physical) disk sharing. More importantly, replication of systems programs in such an uncontrolled fashion tends to cause inconsistent versions of these programs, slow propagation of updates, etc.

2. It is often the case that the code segment of certain system programs can be shared between different invocations of the program, with every invocation having its private data segment. This is obviously not possible between personal computers.

3. In a centralized timesharing system, file system backup is done for the users by a system operator. Since it is often not possible to do file system backups across the network and most personal computers are not outfitted with a tape drive, personal computer users must explicitly transfer their files to a remote file server in order to have them backed up on a regular basis.

4. Finally, while having complete and unlimited access to the resources of his own machine, there is in general no convenient way for a user to make use of the computing power of other processors in the system, even if they are idle. Centralized systems, in contrast, can typically make all of their facilities available to a single user during off hours.

It is sometimes argued that these problems are purely technological in nature, in the sense that their importance will decrease as processors, memory and disks become cheaper and more powerful. For instance, it is argued that, as personal computers become powerful, there will be little need to be able to execute programs on other machines. We see two flaws in this argument. First, it is to be expected that the aspirations of users will become correspondingly larger and therefore will exceed the capabilities of any single machine, however large. Second, while personal computers are good for computing that is personal in nature (such as editing a private file, for instance), some applications inherently do not fit this model very well, in particular those requiring frequent access to shared data. Supporting such applications in a personal computer environment will remain difficult.

Notwithstanding these disadvantages, the personal computer approach has been shown to be workable. In an environment where individual users regard each other with mutual suspicion, this approach is actually quite appropriate because of the inherent autonomy and protection between users. However, the V-System takes an alternative viewpoint. Rather than using the network as a loose connection between the machines (much like a miniature long-haul network), we prefer to view the network as an extended backplane, connecting workstations and servers in a *multiprocessor* arrangement[1]. The V-System then allows the different machines to be integrated into a highly parallel but logically unified computer system, attempting to combine the advantages of personal computers with those of conventional timesharing systems.

This logical unification is achieved by making resource access transparent. This makes the collection of machines appear as a single logical entity to its users. In V, transparent resource access is based on the use of a transparent (message-based) interprocess communication mechanism that provides communication between processes on the same or on different workstations. The rest of the system can then be implemented in approximately the same way as the message-based systems developed for single machines, regardless of the fact that some processes may reside on other machines.

Let us briefly review some of the drawbacks we mentioned about personal computers and see how this approach either solves or at least alleviates some of the problems.

1. Since interprocess communication is transparent, programs can be loaded from a remote (central) file server instead of from the local disk. By extension, no local permanent storage may be necessary at all, thereby significantly reducing the cost of a workstation. System programs are now shared on a single machine, thereby accomplishing some amount of disk sharing. Moreover, the replication problems

[1] The term *multiprocessor* is not intended here to imply the use of shared memory.

alluded to earlier are no longer present. Also, since all user files reside on a centralized file server, file system backups can be done in the traditional way.

2. Some code sharing can occur between programs in the following, non-traditional way. Take again the file system as an example. If every workstation has its own disk and consequently its own file system, no code sharing occurs. However, if there is a single central file server and all workstations use this file server for their secondary storage access, then they effectively share the file server code. Since this code resides on the remote file server machine, this reduces the demands on the processor and the memory of the workstation itself.

3. Programs can be loaded and executed on other workstations, in the same manner as on the local machine, thereby giving users potentially more than one processor at their disposal.

A couple of comments are called for at this point. First, any communication mechanism that is transparent across machine boundaries can accomplish the above goals, whether it is message-based, procedural, based on shared memory or based on some other communication paradigm. Second, all of the above advantages are rendered inconsequential if the communication mechanism does not perform adequately. We concur with Popek et al. [59] that *inefficient transparency is no transparency.* Indeed, if the cost of remote operations is substantially higher than the cost of the corresponding local operations, then the applications programmer has again to distinguish between the two, thereby undoing most of the intended benefits of a transparent mechanism. The V-System's interprocess communication performs sufficiently well to provide efficient transparent access for most applications. The mechanisms for accomplishing such performance form the principal topic of this thesis.

1.2. Overview of the V-System

1.2.1. Hardware Environment

The V-System runs on a set of SUN workstations [6] interconnected by a 3 Mb Ethernet [54] or a 10 Mb Ethernet [27]. The SUN workstation is a 68000-based machine, typical of many workstations currently on the market. It provides (See also Figure 1-1):

1. Approximately 1 MIPS of computing power
2. Up to 2 Megabytes of physical main memory
3. Support for separate virtual address spaces up to 2 Megabyte
4. 1000x1000 bitmap display, corresponding frame buffer with hardware assists for rasterop operations and a mouse
5. 3 Mb Ethernet or 10 Mb Ethernet interface.

Due to the limitations of the 68000 processor, no demand paging is done. The system has been in daily use in this configuration for almost two years now. At the time of writing, the V-System is being ported to the next generation SUN, with a 68010 processor which does allow demand paging. At the same time, implementations on the VAX 11/750 and the IRIS workstation [22] are under way.

1.2.2. Software Environment

The V-System software follows the conventional model of many message-based systems [14, 36, 81]. Logically, it consists of a distributed kernel and a set of server processes, possibly running on dedicated server machines. The distributed kernel itself consists of the collection of kernels resident on each participating machine (See Figure 1-2). The kernel is essentially a *communications server:* it provides communication between processes and very little else. Many functions present in the kernel of other operating systems (file

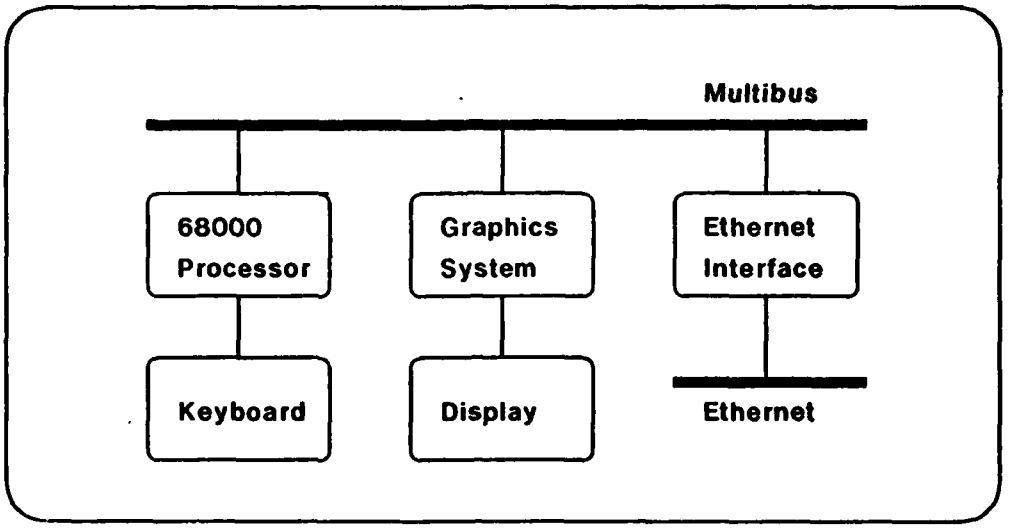


Figure 1-1: SUN Workstation

services e.g.) are performed outside of the kernel in the V-System. Functions other than interprocess communication which require kernel support (process, memory and device management) are provided by a pseudo-process running inside the kernel. Through a low-overhead *interkernel protocol*, the different kernels cooperate to provide transparent communication between processes on different machines. Servers currently include device servers, storage servers, virtual graphics terminal servers and network servers. Device servers are mostly implemented as pseudo-processes in the kernel: in a typical configuration they allow access to the network, the keyboard, the mouse and the graphics frame buffer. All other servers are implemented as (collections of) processes outside of the kernel. The internet server [39] permits its clients to communicate with any host or program accessible via the Xerox PUP [11] or the ARPA Internet protocols [60]. The virtual graphics terminal server [37, 38] supports object-oriented graphics interaction in a device-independent manner. The storage server provides access to a hierarchically structured file system.

The V kernel's interprocess communication has been modeled after that of Thoth [14, 16] and Verex [49], with some influence from DEMOS [5]. Communication between processes is provided in the form of short fixed-length messages, each with an associated reply message, plus a data transfer operation for moving larger amounts of data between processes. The communication primitives promote a *client-server* style interaction. The common communication scenario is as follows (See Figure 1-3): a *client* process executes a ***Send*** to a *server* process, which then completes execution of a ***Receive*** to receive the message and eventually executes a ***Reply*** to respond with a reply message back to the client. The receiver may execute one or more ***MoveTo*** or ***MoveFrom*** data transfer operations between the time the message is received and the time the ***Reply*** message is sent.

A set of standard library routines provides a procedural interface to the message passing primitives. Thus, kernel operations such as ***Send***, ***Receive***, ***Reply***, ***MoveTo*** and ***MoveFrom*** typically appear as procedure calls in user code. The corresponding library routine then invokes the kernel via a trap mechanism. However, many application programs prefer to handle their I/O connections as byte streams rather than having to deal with the low level message interface. In V, a *reliable block stream* is provided by means of the V I/O protocol [13], essentially a set of conventions on the format, the contents and the sequence of messages to be exchanged between clients and servers. Additionally, there are two library packages of interest to this discussion: one that implements a *byte-stream* interface to the V I/O protocol's block stream, and a second that provides essentially the same interface as the Unix C library (See Figure 1-4). Clients typically use the latter library packages, which then perform the appropriate translation into V messages. Servers use the ***Receive*** and ***Reply***

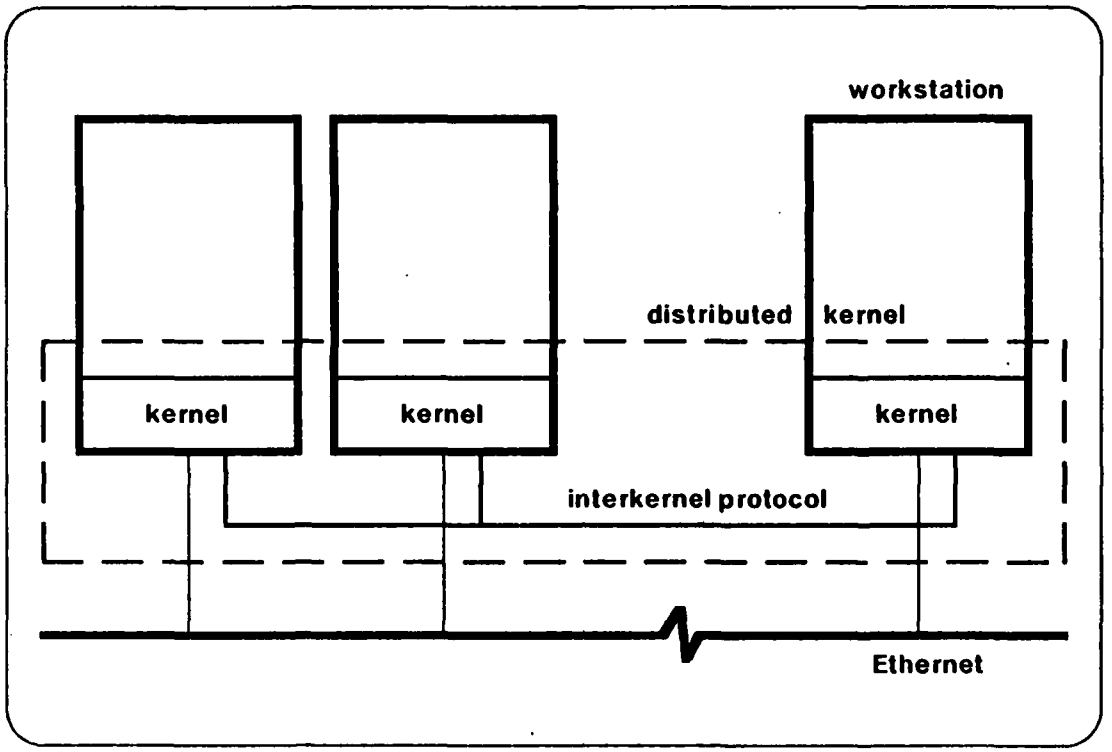


Figure 1-2: The V-System: Overview

kernel primitives directly to implement the I/O protocol. It is essential to recognize that the stream protocol is implemented outside of the kernel. That is, the kernel only transmits the messages between processes, irrespective of their contents or their sequence.

1.3. Thesis Outline

The ideas of this dissertation are developed as follows. Chapter 2 presents a brief overview of related work. Chapter 3 defines the communication primitives available at the kernel interface, and presents the results of an empirical performance evaluation of the kernel. The performance evaluation covers the performance of the kernel by itself (in terms of message passing times), the performance of file access when implemented on top of the V kernel and the performance of other important interprocess communication patterns. Chapter 4 uses data from the performance measurements of Chapter 3 as input to a queueing network model of network page-level file access. The kernel implementation and the interkernel protocol are described in Chapter 5. Performance tradeoffs between various implementation strategies are brought forth and wherever possible quantified. In Chapter 6 we extend the one-to-one interprocess communication of the V kernel to one-to-many communication drawing on the multicast capabilities of many local networks. Finally, in Chapter 7 we summarize the contributions of this thesis and explore avenues for further work.

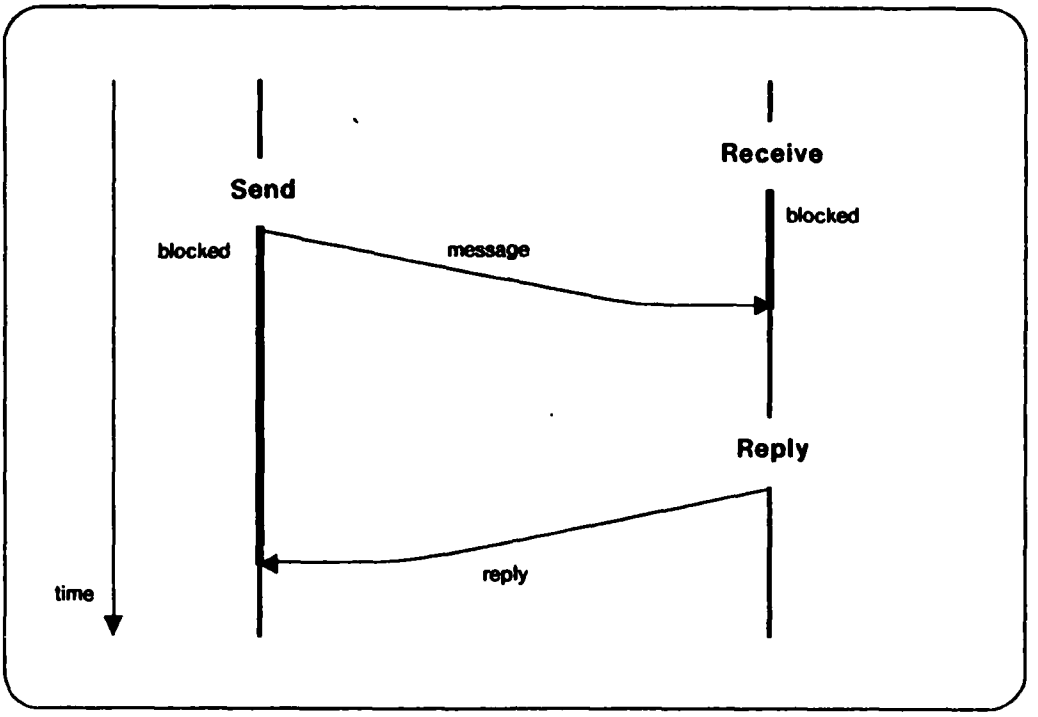


Figure 1-3: ***Send-Receive-Reply*** Message Transaction

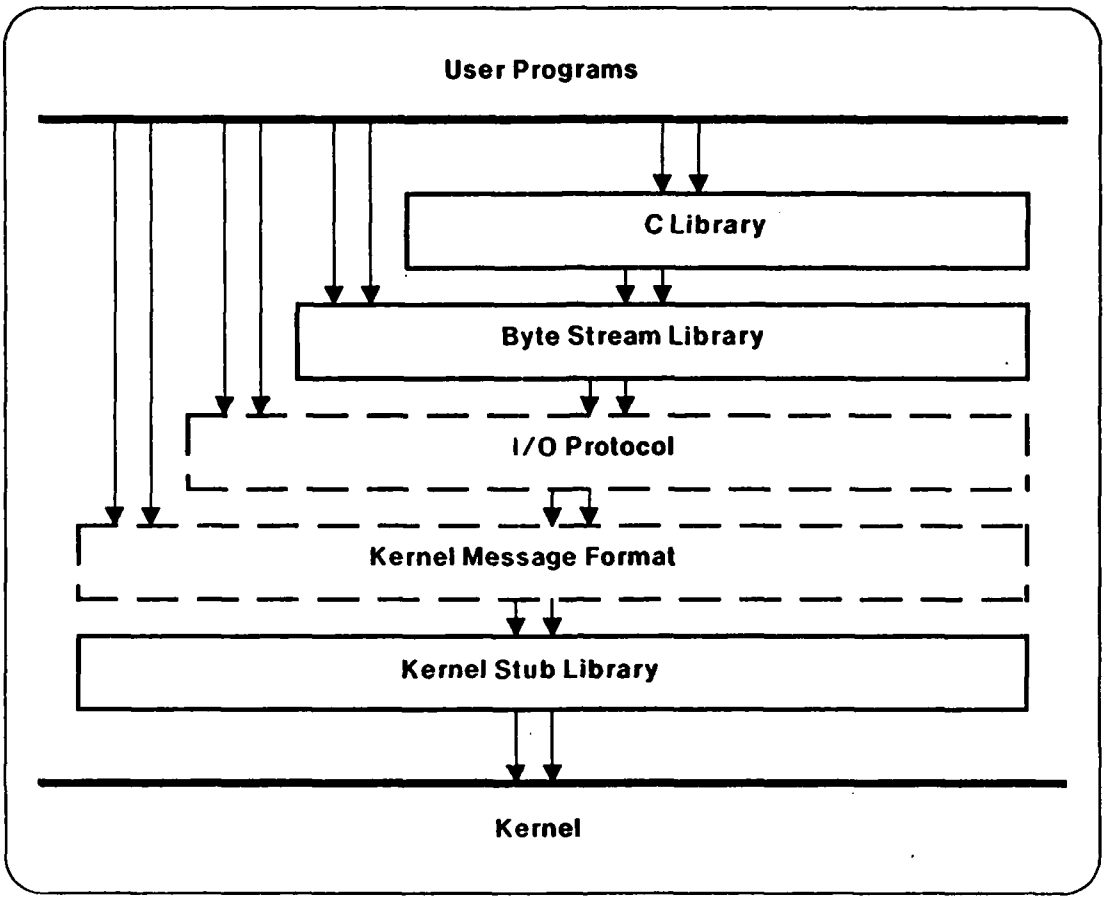


Figure 1-4: Client Interface to the V-System

—2—
Related Work

The purpose of this chapter is to provide the reader with a few representative examples of related efforts, so he can put the work of this thesis in the appropriate context. It is not intended to give an extensive survey of distributed systems research in the last decade. This overview is also restricted to systems that have been implemented and are known to work. The chapter is organized as follows: in each section, a particular *class* of systems is covered. Systems belonging to the class are mentioned and then a prominent example of the class is singled out and used as a vehicle for further discussion. Following is a list of the classes discussed and their flag bearer implementations:

1. Nor-transparent distributed systems: Arpanet
2. Streams and specialized file protocols: LOCUS
3. Enhanced[2] message passing: Accent
4. Reduced message passing: Thoth
5. Remote procedure calls: Cedar RPC
6. Remote memory references: Spector
7. Virtual memory-oriented systems: Apollo DOMAIN
8. Object based systems: Eden

One other system that does not fit any particular class, the Cambridge Distributed System, is also briefly discussed.

2.1. Non-transparent Distributed Systems

The Arpanet [53], the oldest example of a distributed system, is probably also the best example of a non-transparent distributed system. Resources on other machines (such as files or virtual terminals) can only be accessed by different mechanisms than those used for accessing similar resources on the local machine. Although some attempts in the way of transparency have been made [33, 74], too many odds had to be overcome. The long-haul nature of the network, the basic protocol structure, whereby hosts regard each other with mutual suspicion [53], and the heterogeneity of the different machines and operating systems have precluded significant progress in that direction.

Most commercial systems have carried this tradition forward, although they use high-bandwidth, low-delay local network technology. Much of the early Alto software [73] is in this class and also some of the ensuing work on the Xerox Star [31] system. Based on a desire to provide a large degree of autonomy to individual nodes in the network, some of the work done at MIT has also shied away from the notion of transparency [19, 21].

Due to hardware constraints or explicit policy decisions, these systems aim for a much more loosely coupled

[2] We use the terms *enhanced* and *reduced* with respect to message passing in loose analogy to the way they have become known for machine instruction sets.

environment than the V-System. Since they are so different both in scope and motivation, they are not discussed further.

2.2. Streams and Specialized File Access Protocols

Perhaps a more catchy header for this section would be *distributed Unix* systems. Many such systems [50, 62, 66] have been built or are being built, largely because of the popularity of Unix [65] as a single-machine operating system. We use LOCUS [59, 75] as the representative system of this class.

In terms of hardware, the current LOCUS system is geared somewhat more towards a network connecting a number of general purpose multi-user machines while the V-System adheres more to a model of single user workstations and dedicated server machines. The LOCUS kernels on different machines cooperate to give the illusion of system-wide transparency, similar to the distributed V kernel. The LOCUS interkernel protocol is really a collection of protocols, each of which is specialized for a particular application (such as file access, remote tasking, etc.) In contrast, the V interkernel protocol supports interprocess communication. Applications then use interprocess communication as the base layer for building their application-dependent protocols. Of particular interest in the LOCUS interkernel protocol is the *specialized file access protocol*, specifically optimized for high performance sequential file access. In order to further optimize file access, the file system is part of the LOCUS kernel. Although different in the communication model it emphasizes, the LOCUS interkernel protocol is similar to that of the V-System in stripping away protocol layers for increased performance. LOCUS serves as a frequent point of comparison in this thesis, both in terms of the primitives it supports as well as in terms of the system's performance.

2.3. Enhanced Message Passing

Systems in this class all have common ancestry in the single machine DEMOS [5] operating system, developed at Los Alamos for the Cray-1. Among its distributed descendants are Arachne [70] at the University of Wisconsin, SODS [68, 69] at the University of Delaware, DEMOS/MP [61] at Berkeley and Accent [63] at CMU. The latter was developed drawing heavily on the experience gained during the development of the RIG [36] operating system at the University of Rochester and also incorporates several ideas from the Multics file system [26, 58]. It is taken as the representative of this group.

All these systems adhere to the same basic software structuring paradigm as the V-System in that they consist of a communications kernel and a set of server processes that provide other system services. However, the communications interface that these kernels provide is rather sophisticated. Accent, in particular, supports protected message paths, indirect process addresses (ports), asynchronous message communication, arbitrary size messages and structured message formats. The value of providing all these features at the kernel interface has to be weighed against potential penalties in complexity and performance.

2.4. Reduced Message Passing

The seminal system in this class is the Thoth kernel [14, 16]. The goals of the Thoth project, done at the University of Waterloo, were two-fold: to experiment with the multi-process structuring of software and to gain insight in system software portability. Thoth introduced many of the concepts present in V, in particular the idea of synchronous message passing. The Thoth kernel was implemented on a number of machines, including a Nova/2 and a TI-990. Verex [49], a descendant of the original system done at the University of British Columbia, also ran on a master-slave multiprocessor configuration. The V-System interprocess communication has been modeled after that of Thoth, although a number of modifications were made showing an evolution to a more sophisticated kernel interface.

2.5. Remote Memory References

We take Spector's thesis work as a representative sample of this category [71, 72]. Another example is the work done by Leblanc [45, 46] in the context of the StarMod programming language [23], although his work includes many other models of interprocess communication.

Spector's work again attempts to provide the same sort of integration as the V-System. Although similar in its goals, Spector proposes a fundamentally different interconnection mechanism, known as the *remote memory reference* model. All machines in the system share a common address space and any machine can transparently make memory accesses to any location in that address space, regardless of which machine the actual physical memory location resides on. Again, as in LOCUS and in V, layering of protocols is avoided for performance reasons. Both Spector's and Leblanc's theses give extensive performance measurements of their respective communication primitives.

2.6. Virtual Memory-Oriented Systems

The representative system in this class is the Apollo DOMAIN system [44]. The distinctive features of the DOMAIN include: a network-wide file system of objects addressed by unique identifiers (UIDs); a single-level store for transparently accessing all objects, regardless of their location in the network; and a network-wide hierarchical name space. A unique aspect of the DOMAIN system is its network-wide single-level store (Multics [26] is an example of a single-level store for a centralized system.) Programs access objects by presenting their UIDs and asking them to be mapped into their address space. Subsequently, they are accessed with ordinary machine instructions, utilizing virtual memory demand paging. The purpose of the single-level store is not, unlike the systems referred to in the previous section, to create network-wide shared memory semantics akin to that provided by a closely coupled multiprocessor. Instead, it is a form of lazy evaluation: only required portions of objects are retrieved from disk or over the network. Another purpose is to provide a uniform, network transparent way to access all objects: the mapping operation is independent of whether the UID is for a local or remote object. We note that, as an alternative to its single-level store, the DOMAIN system also supports network transparent interprocess communication through so called *sockets*.

2.7. Remote Procedure Calls

The idea of remote procedure calls is quite appealing: procedure calls are the predominant mechanism for transfer of control between programs or program modules on a single machine. It is simply proposed that their semantics be extended across machine boundaries. The first mention of remote procedure calls dates back to 1976 and was made in the context of the Arpanet network [77]. Their applicability to local network environments was first explored in Nelson's thesis [57]. Nelson suggested various semantic definitions and explored the performance tradeoffs of different implementation strategies. The first full-fledged implementation was carried out by Birrell and Nelson in the framework of the Cedar programming environment at Xerox PARC [9]. Their implementation runs on Dorados [35] interconnected by a 3 Mb Ethernet [54]. The Cedar RPC implementation again depends heavily on specialized protocols for adequate performance. Wherever possible, we compare its performance with the measurements obtained for the V-System.

2.8. Object-Based Systems

The object model has long been a popular program structuring tool. It has its roots as far back as the Simula language [24]. Among the object based distributed systems of interest are Eden [3, 41], Argus [47, 48] and Clouds [2] (While these are "pure" object systems, virtually all server-based systems can be considered variants of the object model.)

The Eden project addresses the construction of an environment similar to that of the V-System. Its

emphasis is, however, much more on programmability than on performance. The Eden world consists of a virtual space of typed objects, possibly residing on different machines and communicating via a fairly heavy-handed *invocation* mechanism. Within the Eden objects, multiple concurrent processes can coexist. The system is currently implemented on a guest-level to Unix 4.1BSD. Argus and Clouds are similar to Eden in philosophy. Unlike Eden though, they both support atomic transactions on objects at the kernel level.

2.9. The Cambridge Distributed System

The Cambridge distributed system model [56, 78] also takes the view of a local network as an extended backplane between machines. The local network in their case is the Cambridge ring [79]. There are dedicated servers on the ring, but there is no explicit idea of personal workstations, at least in the traditional sense. Instead there is a *processor bank*, consisting of a number of processors that are in principle available to any user. When a user then logs in to the network (through a terminal concentrator connected to the net), he gets allocated a processor out of the processor bank. If needed and if extra processors are available, more processors can be allocated to a single user.

2.10. Chapter Summary

Except for the Arpanet, all systems surveyed in this chapter share the goal of the V-System to create a single logical entity out of several loosely connected machines. A number of systems have specifically addressed the problem of high-performance communication, in particular the LOCUS system and Cedar RPC. LOCUS achieves high performance by the use of protocols specifically tuned to certain applications, in particular file access. In contrast, the V-System provides a single set of interprocess communication primitives and a single supporting protocol, which is then used by applications as a base layer for structuring their application-specific communication needs. The question naturally arises whether this interposition of an extra layer does not compromise the performance of certain key applications such as file access. This question is addressed in the next chapter, where we present the V interprocess communication primitives and their performance. The RPC model presents a similar approach to that of the V-System in presenting a single set of primitives to applications rather than application-dependent primitives and protocols. However, current work, in particular Nelson's thesis [57], has been more concerned with the effects on performance of implementing the RPC protocol at various layers of a protocol hierarchy rather than with the performance of applications implemented on top of RPC.

— 3 —
The V Kernel Primitives and their Performance

3.1. Introduction

So far, we have described in general terms the research area and the V-System, which functions as the concrete research setting. The goal of the V-System is the construction of an integrated distributed system, whereby host boundaries are hidden from the user. We have indicated how the design of an efficient transparent interprocess communication mechanism -- the topic of this thesis -- is an important step towards achieving that goal. In this chapter we define the interprocess communication facilities available at the V kernel interface and we present the results of an empirical performance evaluation of the kernel (A preliminary version of this chapter appeared in [17].) The figures in this chapter are in a sense raw numbers; they are obtained by conducting experiments in a given hardware environment and in an otherwise idle system. In the next chapter we use these raw numbers as inputs to the construction of a queueing network model of network page-level file access. That way we can assess the effects of changing hardware parameters or increasing load on the performance measurements presented in this chapter.

This chapter is organized as follows: Section 3.2 contains the definition of the V kernel interprocess communication primitives. Section 3.3 shows the application of these primitives in a simple example. In Section 3.4, we describe our measurement methods and we characterize the hardware environment in which the experiments took place. Next, in Section 3.5, we give performance measurements of individual kernel primitives. Finally, the bulk of this chapter is taken up by Sections 3.6 and 3.7 in which we discuss performance figures for two important applications built on top of the V kernel, namely file access and pipe data transfer.

3.2. The V Kernel Interprocess Communication Primitives

In V, processes are identified by a 32-bit unique identifier, called the *process identifier*. They communicate via the synchronous exchange of messages. Every request has a reply associated with it and the sender of a message blocks until the message is received and replied to. Messages are normally small and fixed-size (8 32-bit words) although under some circumstances a segment of data can be appended to a message. This segment is variable in size up to a fixed maximum. There is a separate data transfer facility for moving larger amounts of data. Additionally, there are primitives for low-level name registration and lookup, and for querying the existence of a process. The complete set of interprocess communication facilities follows[3].

Send (message, processId)

Send the message in ***message*** to the process with process identifier ***processId***, blocking the active process until the message is both received and replied to.

The kernel legislates a message format, by which a process specifies in the message the *segment* of its address space that the message recipient may access and whether the recipient may read or write that segment. A segment is specified by the last two words of a message, giving its start address and its length respectively. Reserved flag bits at the beginning of the message indicate whether a segment is specified and if so, its access permissions.

[3] The definitions of the primitives in this chapter are taken from [7] with some omissions for the sake of brevity.

It is intended and assumed that most logical requests can be assigned a request code that is stored in the first word of the request message so that the bits are set correctly for the request by the value of the request code.

(processId, byteCount) = Receive (message, segmentPointer, segmentSize)

Suspend the active process until a message is available from a sending process. Return the ***processId*** of that process, leave the message in the array pointed to by ***message*** and at most the first ***segmentSize*** bytes of the segment included with the message in the receiver's address space starting at ***segmentPointer***. The actual number of bytes received in the segment is returned in ***byteCount***. Messages are queued in first come, first served order.

processId2 = ReceiveSpecific (message, processId1)

Suspend the active process until a message is available from the process specified by ***processId1***, returning the process identifier of this process in ***processId2*** and the message in the array pointed to by ***message***. If ***processId1*** is not a valid process identifier, ***ReceiveSpecific*** returns 0. ***ReceiveSpecific*** is almost exclusively used in the latter fashion, to query the existence of process ***processId1***.

Reply (message, processId, destinationPointer, segmentPointer, segmentSize)

Send the reply message specified by ***message*** and the segment beginning at ***segmentPointer*** of length ***segmentSize*** to the process specified by ***processId***. The specified process must be awaiting reply from the active process, and if the segment size is different from zero, the appropriate access rights must have been given as described under ***Send***. The reply message overwrites the original message in the sender and the segment, if present, is placed at ***destinationPointer*** in the sender's address space.

Forward (message, processId1, processId2)

Forward the message pointed to by ***message*** to the process specified by ***processId2*** as though it had been sent by the process ***processId1***. ***processId1*** must be awaiting reply from the active process. The effect of this operation is the same as ***processId1*** sending directly to ***processId2*** except for the active process being noted as the forwarder of the message. Note that ***Forward*** does not block.

MoveFrom (sourcePid, destinationPointer, sourcePointer, byteCount)

Copy ***byteCount*** bytes from the segment starting at ***sourcePointer*** in the address space of ***sourcePid*** to the segment starting at ***destinationPointer*** in the active process's address space. The ***sourcePid*** process must be awaiting reply from the active process and must have provided read access to the appropriate segment of its address space using the message format conventions described under ***Send***.

MoveTo (destinationPid, destinationPointer, sourcePointer, byteCount)

Copy ***byteCount*** bytes from the segment starting at ***sourcePointer*** in the active process's address space to the segment starting at ***destinationPointer*** in the address space of the ***destinationPid*** process. The ***destinationPid*** process must be awaiting reply from the active process and must have provided write access to the appropriate segment of its address space using the message format conventions described under ***Send***.

SetPid (logicalId, processId, scope)

Associate ***processId*** with the specified ***logicalId*** within the specified ***scope***. Subsequent calls to ***GetPid*** with this ***logicalId*** and appropriate ***scope*** return this ***processId***. The ***scope*** is one of ***local***, ***remote*** or ***both***.

processId = GetPid (logicalId, scope)

Return the ***processId*** of the process registered using ***SetPid*** with the specified ***logicalId*** and ***scope***, 0 if not set. The ***scope*** is one of ***local***, ***remote*** or ***both***.

3.3. Example of Use

At this point an example of use is in order. Readers familiar with the concepts of the V-System can easily skip this section and proceed immediately to Section 3.4. The example uses a loose variation of the C programming language [34]. While the example is kept simple for the sake of brevity, we show how it can be extended to more realistic applications.

3.3.1. Specification and Implementation

We wish to implement a server process that responds to two client requests, ***GetData*** and ***PutData***. The server has a single data buffer, from which the data is to come on a ***GetData***, and into which the data is to be placed on a ***PutData***. Clients are expected to specify in their requests the size of the data transfer they expect to happen. Thus, on a ***PutData*** with a certain size, the buffer is filled up to that size with the corresponding data. The contents of the rest of the buffer is undefined. On a subsequent ***GetData***, the contents of the buffer is returned up to the size requested in the ***GetData***, or up to the size of the preceding ***PutData***, whichever is smaller.

Figure 3-1 lists some type definitions that form the interface between client and server. Figures 3-2 and 3-3 show the server's main routine and some auxiliary routines, respectively. A client executing a ***PutData*** followed by a ***GetData*** is shown in Figure 3-4. A discussion follows in Section 3.3.2.

```
/* Client - Server Interface */

RequestCode     = enum ( getData, putData );
ReplyCode       = enum ( requestTooBig, notImplemented, OK );

/* Format of the messages between client and server */

Request         = record
                    {
                      RequestCode       requestcode;
                      ...
                      char              *bufferPtr;
                      unsigned          bufferSize;
                    }

Reply           = record
                    {
                      ReplyCode         replycode;
                      ...
                      unsigned          bufferSize;
                    }
```

Figure 3-1: Client - Server Interface

3.3.2. Discussion

3.3.2.1 Server Implementation

Figure 3-2 shows a typical construction for a simple server in the V-System. First, the server initializes its data structures (In this simple case this is only the variable ***currentBufferSize***), then announces itself by executing a ***SetPid***. Then it goes into a loop waiting for incoming messages. Depending on the request code in an incoming message, appropriate action is taken to satisfy this request. When the incoming request is a ***GetData***, the server uses a ***MoveTo*** to move the data from its buffer to the client's buffer. For a ***PutData***, a ***MoveFrom*** accomplishes the data transfer in the other direction. Finally, the server puts a reply code in the reply message, and executes a ***Reply***.

3.3.2.2 Client Implementation

The client first finds out the process identifier of the server by executing a ***GetPid*** (See Figure 3-4). It then formats the appropriate messages and sends them off to the server. The exact value of the ***GetData*** and ***PutData*** request codes is not shown here. Appropriate bits in both request codes have to be set such that the client provides access to the segment of its address space containing the data buffer, read access in the case of ***PutData*** and write access in the case of ***GetData***. This is necessary to allow the server's ***MoveTo*** and ***MoveFrom*** operations to succeed.

In practice, clients would seldom use the message passing primitives explicitly, as suggested in this example. Typically, with each server there exists a set of library routines, which provide a procedural interface to the server's message interface. Thus, clients would call such a library routine, which then takes care of formatting the message and sending it to the server.

3.3.2.3 More Sophisticated Server Implementation

A more realistic implementation of a server process, for instance a file server, would have to take into account a number of additional considerations. First, data is not always available immediately on a ***GetData*** request. In many cases, it has to be fetched from the disk first. Second, it is unlikely that the server has sufficient buffer space to read the whole file into its buffer and then transfer it to the client via a ***MoveTo***. A more realistic server strategy would be to allocate a fixed-size buffer in which it could fetch a block from the disk, then transfer that block to the client, fetch another block in the same buffer, and so forth. In that case, the ***GetData*** routine would look as in Figure 3-5.

Finally, it is often the case that a small amount of data is to be transferred between client and server, for instance a single file page. Such applications would take advantage of the capability to receive such a small segment together with a message, or to add a small segment to a ***Reply*** message. For instance, the ***GetData*** routine would look as in Figure 3-6.

This completes the description of the V interprocess communication primitives. For more detail, the reader is referred to [7]. We now turn to the performance evaluation of the kernel. First, we describe in detail the experimental environment in which the measurements were conducted and the measurement methods used.

3.4. The Experimental Environment

3.4.1. Measurement Methods

Measurements of individual operations are performed by executing the operation N times (typically 1000 times), recording the total time required, subtracting loop overhead and other artifact, and then dividing the total time by N. Measurement of total time relies on the software maintained V kernel time which is accurate plus or minus 10 milliseconds.

Measurement of processor utilization is done using a low priority "busywork" process on each workstation

```
DataServer()
  {
    char        Buffer[maxBufferSize];  /* Data buffer */
    ProcessId   pid;                    /* Requester */
    Message     msg;                    /* Message buffer */
    Request     *req = (Request *) msg; /* Buffer for request */
    Reply       *rep = (Reply *) msg;   /* Buffer for replies */
    ReplyCode   reply;                  /* Reply code */
    unsigned    currentBufferSize;      /* Current buffersize */

    /* Initialize */
    currentBufferSize = 0;
    /* Register as the data server */
    SetPid( dataServer, GetPid( activeProcess, local), any );

    /* Wait for requests to come in */
    while( true )
      {
        ( pid, size ) = Receive( msg, NULL, 0 );
        switch( req->requestcode )
          {
            case putData:
              {
                reply = PutData( pid, req );
                break;
              }
            case getData:
              {
                reply = GetData( pid, req );
                break;
              }
            default:
              {
                reply = notImplemented;
                break;
              }
          }
        rep->replycode = reply;
        Reply( msg, pid, NULL, NULL, 0 );
      }
  }
```

Figure 3-2: Server: Main Routine

```
ReplyCode PutData( pid, req ) ProcessId pid; Request *req;
  {
    extern char         Buffer[];
    extern unsigned     currentBufferSize;

    if( req->bufferSize > maxBufferSize )
        return( requestTooBig );

    MoveFrom( pid, Buffer, req->bufferPtr, req->bufferSize );

    currentBufferSize = req->bufferSize;

    return( OK );
  }

ReplyCode GetData( pid, req ) ProcessId pid; Request *req;
  {
    extern char         Buffer[];
    extern unsigned     currentBufferSize;

    if( req->bufferSize > currentBufferSize )
        req->bufferSize = currentBufferSize;

    MoveTo( pid, req->bufferPtr, Buffer, req->bufferSize );

    return( OK );
  }
```

Figure 3-3: Server: Auxiliary Routines

```
Client()
  {
    ProcessId           serverPid;
    Message             msg;
    Request             *req = (Request *) msg;
    unsigned            myBufferSize;
    char                *myBufferPtr;

    /* Allocate a buffer */

    myBufferPtr = malloc( 1, myBufferSize );

    /* Locate the server */

    serverPid = GetPid( dataServer, any );

    /* Do a PutData */

    req->requestcode = putData;
    req->bufferPtr = myBufferPtr;
    req->bufferSize = myBufferSize;

    Send( req, serverPid );

    /* Do a GetData */

    req->requestcode = getData;
    req->bufferPtr = myBufferPtr;
    req->bufferSize = myBufferSize;

    Send( req, serverPid );
  }
```

Figure 3-4: Client

that repeatedly updates a counter in an infinite loop. All other processor utilization reduces the processor

```
ReplyCode GetData( pid, req ) ProcessId pid; Request *req;
  {
    extern char         Buffer[];
    unsigned            i, numBlocks;

    /* Assume requests are multiples of block size */

    numBlocks = req->bufferSize / blockSize;

    /* Transfer data, block by block */

    for( i=0; i<numBlocks; i++ )
      {
        ReadDataOffDisk( Buffer, blockSize );
        MoveTo( pid, req->bufferPtr + i*blockSize,
                                      Buffer, blockSize );
      }
    return( OK );
  }
```

Figure 3-5: Using Multiple *MoveTo* operations

```
ReplyCode GetData( pid, req ) ProcessId pid; Request *req;
  {
    extern char         Buffer[];

    ReadDataOffDisk( Buffer, pageSize );
    Reply( req, pid, req->bufferPtr, Buffer, pageSize );
  }
```

Figure 3-6: Appending Small Segments to Messages

allocation to this process. Thus, the processor time used per operation on a workstation is the total time minus the processor time allocated to the "busywork" process divided by N, the number of operations executed.

Using 1000 trials per operation and time accurate to plus or minus 10 milliseconds, our results should be accurate to about 0.02 milliseconds except for the effect of variation in network load. These variations were observed to be minimal under normal circumstances (i.e. when no packets are lost). Packet loss can cause a single exchange to take substantially longer, but its low frequency makes the effect invisible in the calculation of the mean values.

3.4.2. Hardware Environment

The hardware environment in which the experiments are to take place was described earlier (See Section 1.2.1). In summary, the system runs on 68000-based SUN workstations, running different clock speeds (8 and 10 Mhz), the latter being approximately a 1 MIPS machine. Two networks are used, a 3 Mb and a 10 Mb Ethernet. However, neither the processor speed nor the network data rate fully capture the potential of a particular machine-network configuration with respect to network access. In the next section, we introduce the notion of *network penalty*, which provides a better characterization of a hardware configuration in this respect.

3.4.3. Network Penalty

Our measurements of the V kernel are primarily concerned with two comparisons:

- The cost of remote operations versus the cost of the corresponding local operations.
- The cost of remote access using V kernel remote operations versus the cost for other means of remote access.

An important factor in both comparisons is the cost imposed by network communication. In the second

comparison, the basic cost of moving data across the network is a lower bound on the cost for any network access method. In the first comparison, the cost of a remote operation should ideally be the cost of the local operation plus the cost of moving data across the network (data that is in shared kernel memory in the local case) minus the amount of concurrency that can be achieved when the operation is executed on two different machines. For instance, a local message ***Send*** passes pointers to shared memory buffers and descriptors in the kernel while a remote message ***Send*** must move the same data across the network. On the other hand, some work on the sending machine (like blocking the sending process and performing a process switch, if necessary) can be done in parallel with the receiving machine copying the message packet out of the network interface and activating the receiving process. This argument also assumes that the necessary provisions for remote communication do not noticeably degrade the performance of local communication. Although not shown here, this assumption is correct for the V kernel.

To quantify the cost of network communication, we define a measure we call the *network penalty*. The network penalty is defined to be the time to transfer N bytes from one workstation to another in a network datagram on an idle network and assuming no errors. The network penalty is a function of the processor, the network, the network interface and the number of bytes transferred. It is the minimal time penalty for interposing the network between two software modules that could otherwise transmit the data by passing pointers. The network penalty is obtained by measuring the time to transmit N bytes from the main memory of one workstation to the main memory of another and vice versa, and dividing the elapsed time for the experiment by 2. The transfers are implemented at the data link layer and interrupt level so that no protocol or process switching overhead appears in the results. The measurement results therefore provide an accurate assessment of the potential of a particular processor-network configuration with respect to network communication.

Let us first consider data transfers that fit within a single network packet[4]. Measurements of network penalty were made using the 3 Mb Ethernet and the 10 Mb Ethernet. In all measurements, the network was essentially idle due to the unsociable times at which measurements were made. The assumption of an idle network is consistent with the utilization of most local networks. For instance, our network averages 1 to 2 percent utilization. Table 3-1 lists our measurements of the 3 Mb network penalty for the SUN workstation using the 8 and 10 MHz processors, and Table 3-2 provides the results for a 10 Mb Ethernet, using the 3-Com interface [1]. The network time column gives the time for the data to be transmitted based on the physical bit rate of the medium, namely 3 or 10 Mb.

Single Packet Network Penalty -- 3 Mb

Bytes	Network Time	Network Penalty	
		8 MHz	10 MHz
64	0.17	0.80	0.65
128	0.35	1.20	0.96
256	0.70	2.00	1.62
512	1.39	3.65	3.00
1024	2.78	6.95	5.83

Table 3-1: 3 Mb Ethernet SUN Workstation Network Penalty (times in msec.)

The difference between the network time, computed at the network data rate, and the measured network penalty time is accounted for primarily by the processor time to generate and transmit the packet and then receive the packet at the other end. For instance, for a 1024 byte packet, using an 8 MHz processor and the 3 Mb network, the copy time from memory to the Ethernet interface and vice versa is roughly 1.90 milliseconds in each direction. Thus, of the total 6.95 milliseconds network penalty on the 3 Mb network, 3.80 is copy time, 2.78 is network transmission time and 0.3 is (presumably) network and interface latency. If we consider

[4] A number of practical considerations impose a maximum packet size of about 1024 bytes on our 3 Mb network. The maximum packet size on the 10 Mb Ethernet is 1536 bytes.

Single Packet Network Penalty -- 10 Mb

Bytes	Network Time	Network Penalty	
		8 MHz	10 MHz
64	0.05	0.73	0.55
128	0.10	1.00	0.80
256	0.20	1.63	1.29
512	0.41	2.86	2.29
1024	0.82	5.13	4.26

Table 3-2: 10 Mb Ethernet SUN Workstation Network Penalty (times in msec.)

a 10 Mb Ethernet, a similar argument holds, making the processor time approximately 75 percent of the overall network penalty. The importance of the processor speed is also illustrated by the difference in network penalty for the two processors measured in Tables 3-1 and 3-2.

Let us now consider the case where the data transfer requires multiple packets to be sent from the sender to the receiver. The naive approach for obtaining the network penalty for such a multi-packet data transfer would be to sum the network penalties for the individual packets. If one wished to experimentally measure this quantity, for instance for a 4-packet transfer, an experiment would be set up as depicted on the left hand side of Figure 3-7 (labeled "non-streamed transfer"). A packet is sent from the sender to the receiver; a packet is returned from the receiver to the sender; and this procedure is repeated 4 times. The elapsed time is measured and divided by two in order to obtain the network penalty.

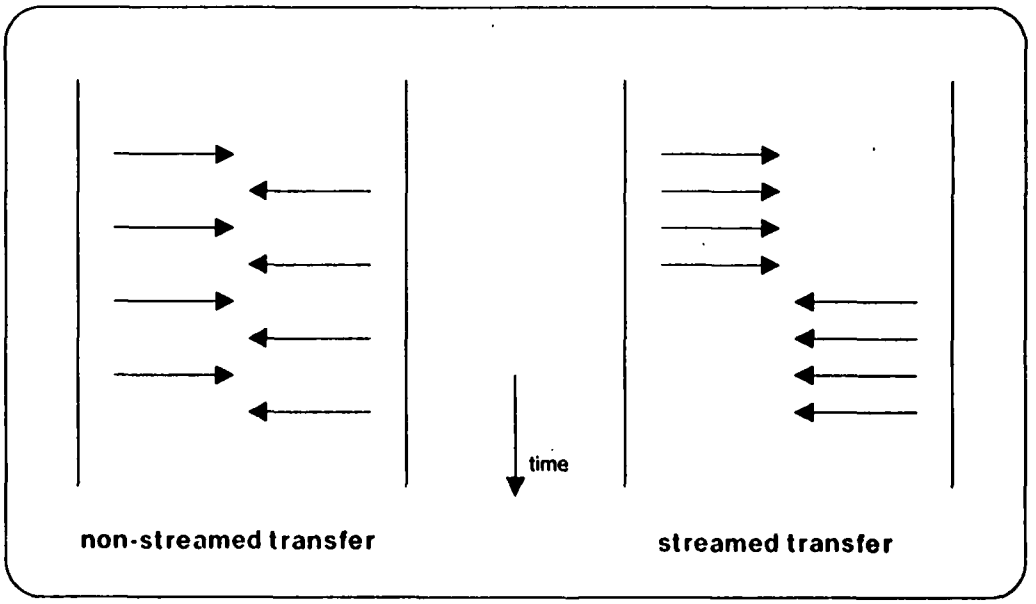


Figure 3-7: Packet Traffic for Network Penalty Measurements
for Non-Streamed and Streamed Multi-Packet Transfers

This is however not the implementation of choice for multi-packet transfers on a local-area network. Due to the low latency of such a network, multi-packet transfers are implemented more efficiently in "streamed

mode", as suggested by the right hand side of Figure 3-7. All necessary packets are transferred from the sender to the receiver and then an equal number of packets is returned from the receiver to the sender. When the elapsed time for this operation is measured and divided by two, a significantly lower value for the network penalty is obtained. The reason for this difference is explained in Figure 3-8. The top line in this figure corresponds to the transfer in non-streamed mode, while the bottom line corresponds to the streamed transfer. The time axis runs horizontally from left to right and the example is for the case of the transfer requiring two packets in each direction.

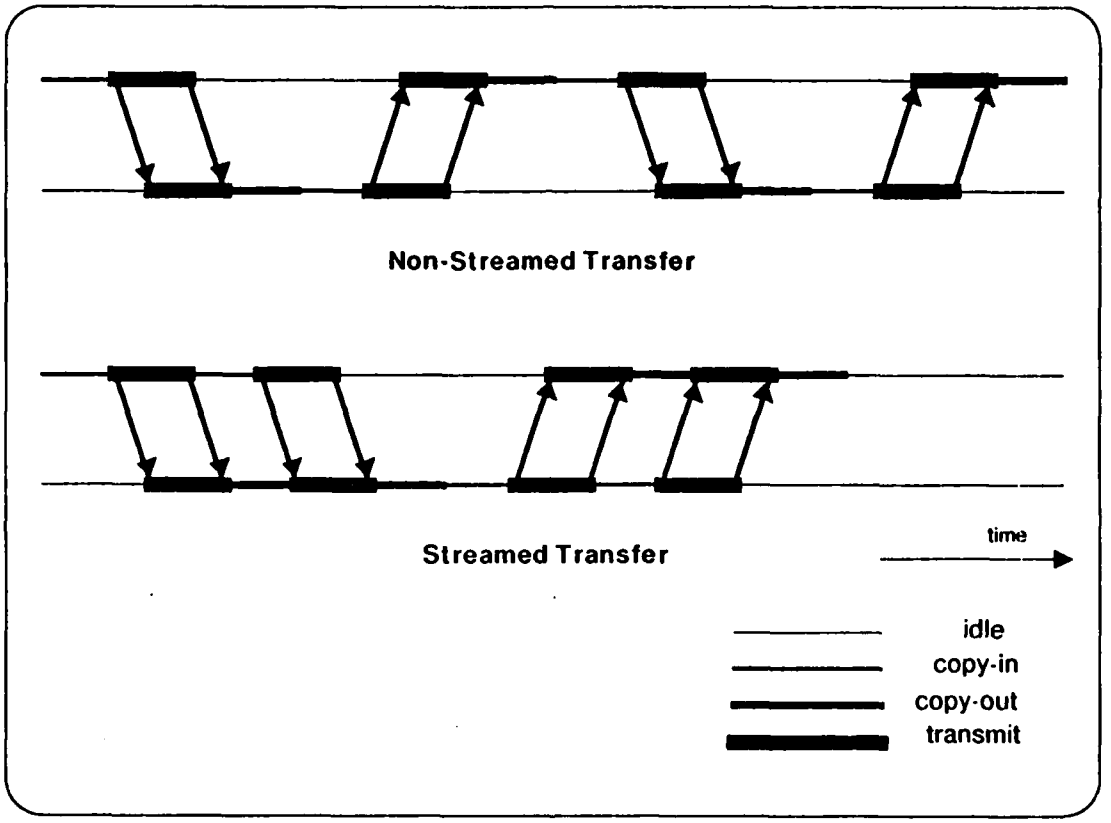


Figure 3-8: Advantages of Streamed Multi-Packet Transfers

Consider first the chronological sequence of events in the case of non-streamed transfer. The sending processor copies a packet from main memory to its interface and then the interface puts the packet on the network. After a time period equal to the network propagation delay the packet arrives at the receiver's interface and then it is copied from the receiver's interface into the receiver's memory by the receiving processor. This process repeats itself in the reverse direction for the packet that is returned from the receiver to the sender, and then again for the next packet, and so forth. Note that the two processors are never active in parallel. This is not the case when the transfer is done in streamed mode, as shown on the second line of Figure 3-8. Due to the very low propagation delay of a local network, the packet is received in the receiver's interface almost completely concurrently with the sender's interface transferring it over the network[5]. Therefore the processor on the sending machine can start copying the next packet from memory to the

[5] In fact, the propagation delay is far exaggerated in Figure 3-8 to make it visible at all: typical propagation delays on a local network are on the order of 10 microseconds while the copy and transmission times depicted in Figure 3-8 are on the order of 1 millisecond.

interface in parallel with the processor on the receiving machine copying the previous packet out of its interface into its memory. Due to the fact that these copies happen in parallel, and, as we saw before, the copy times contribute significantly towards the overall elapsed time, streamed transfer results in values of the network penalty that are substantially lower than those obtained for non-streamed transfers. Measurement results for the network penalty of multi-packet transfers in streamed mode are reported in Table 3-3 and 3-4. The network penalty for non-streamed transfer can be computed by simply multiplying the figures in Table 3-1 and 3-2 by the appropriate factors.

Multi-Packet Network Penalty

Bytes	Network Time	Network Penalty	
		8 MHz	10 MHz
2048	5.57	11.64	9.96
4096	11.14	21.75	18.95

Table 3-3: 3 Mb Ethernet SUN Workstation Network Penalty (times in msec.)

Bytes	Network Time	Network Penalty	
		8 MHz	10 MHz
2048	1.64	7.72	6.04
4096	3.28	13.90	11.11

Table 3-4: 10 Mb Ethernet SUN Workstation Network Penalty (times in msec.)

Using the 10 Mhz processor and 4 kilobytes per data transfer, these figures indicate effective data rates of 1.7 Mb and 3.4 Mb respectively, far below the advertised network data rates of 3 Mb and 10 Mb. Even if we were to let the size of the data transfer become arbitrarily large, the effective data rate would still be limited by the inverse of the sum of the packet copy time plus the packet transmission time, and therefore be noticeably inferior to the network data rate. Higher throughput values can be achieved if the interface provides double buffering. In that case, if the processor can fill the buffer at least as fast as the interface can write it to the network, the interface will be permanently busy, achieving, at least in theory, a throughput comparable to the data rate of the network. In practice, the throughput would probably be somewhat lower due to device latency and network contention.

3.5. Interprocess Communication Performance

Our first set of kernel measurements focuses on the speed of local and network interprocess communication. The kernel performance is presented in terms of the times for message exchanges and the data transfer operations.

3.5.1. Kernel Measurements

Table 3-5 gives the results of our measurements of message exchanges and data transfer with the kernel running on workstations using an 8 MHz processor and connected by a 3 Mb Ethernet. The columns labeled *Local* and *Remote* give the elapsed times for these operations executed locally and remotely. The *Difference* column lists the time difference between the local and remote operations. The *Penalty* column gives the network penalty for the amount of data transmitted as part of the remote operation. The *Client* and *Server* columns list the processor time used for the operations on the two machines involved in the remote execution of the operation. Table 3-6 gives the same measurements using a 10 MHz processor and Tables 3-7 and 3-8 refer to measurements with the 10 Mb network and the same processors. The times for both processors are given to indicate the effect the processor speed has on local and remote operation performance. As expected, the times for local operations, being dependent only on the processor speed, are 25 percent faster on

Kernel Performance

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
	Local	Remote	Diff.		Client	Server
Send-Receive-Reply	1.13	3.18	2.05	1.60	1.79	2.30
MoveFrom: 1 Kbyte	1.35	9.03	7.68	8.15	3.76	5.69
MoveTo: 1 Kbyte	1.34	9.05	7.71	8.15	3.59	5.87

Table 3-5: Kernel: 3 Mb Ethernet and 8 MHz Processor (times in msec.)

Send-Receive-Reply	0.94	2.54	1.60	1.30	1.44	1.79
MoveFrom: 1 Kbyte	1.19	8.00	6.81	6.77	3.32	4.78
MoveTo: 1 Kbyte	1.19	8.00	6.81	6.77	3.17	4.95

Table 3-6: Kernel: 3 Mb Ethernet and 10 MHz Processor (times in msec.)

Send-Receive-Reply	1.13	2.68	1.55	1.46	1.59	2.13
MoveFrom: 1 Kbyte	1.35	6.52	5.17	6.27	3.10	4.68
MoveTo: 1 Kbyte	1.34	6.51	5.17	6.27	3.19	4.67

Table 3-7: Kernel: 10 Mb Ethernet and 8 MHz Processor (times in msec.)

Send-Receive-Reply	0.94	2.23	1.29	1.10	1.30	1.74
MoveFrom: 1 Kbyte	1.19	5.83	4.64	5.16	2.38	4.04
MoveTo: 1 Kbyte	1.19	5.86	4.67	5.16	2.49	4.04

Table 3-8: Kernel: 10 Mb Ethernet and 10 MHz Processor (times in msec.)

the 25 percent faster processor. The almost 15 percent improvement for remote operations indicates the processor speed is a significant performance factor for remote communication and is not rendered insignificant by the network delay.

3.5.2. Interpreting the Measurements

A number of interesting observations can be made based on these figures. First, it can be seen that a significant level of concurrent execution takes place between workstations even though the message-passing is fully synchronized. For instance, transmitting the packet, blocking the sender and switching to another process on the sending workstation proceeds in parallel with the reception of the packet and the readying of the receiving process on the receiving workstation. Concurrent execution is indicated by the fact that the total of the server and client processor times is greater than the elapsed time for a remote message exchange (See the *Client* and *Server* columns in the above tables). In particular, for the 10 Mb network and the 10 Mhz processor, the elapsed time of 2.23 milliseconds has to be compared to the sum of the client and server processor utilizations which amounts to 3.04 milliseconds, indicating that both processors are concurrently active during 36 percent of the elapsed time.

Second, we have argued before that a good lower bound on the remote message time is the sum of the local message time and the network penalty minus the concurrency observed between the two processors. In fact, the remote message time is about 1 millisecond higher than this lower bound (2.2 milliseconds vs. a lower bound of 1.2 milliseconds). This discrepancy is not unexpected and is primarily accounted for by the need to dynamically allocate a buffer for the incoming message on the receiving machine. Specifically, for local message passing, no dynamic buffer allocation is necessary because the message can be buffered in the process descriptor of the sender. When a message comes in over the network, a buffer has to be allocated dynamically. Similarly, when the ***Reply*** is sent, that buffer has to be deallocated. A newer version of the kernel has at all times such a buffer preallocated in order to reduce this cost. Additional costs that are incurred on top of those accounted for in the calculation of the lower bound are the need for locating the host of the remote process and the fact that in the kernel the Ethernet device is shared between the kernel and application programs.

As a final observation, we claim that the absolute difference between local and remote message times is sufficiently small for it to be possible to intermix local and remote message exchanges rather freely. Some care is required in interpreting this statement. Superficially, the fact that the remote ***Send-Receive-Reply*** sequence (for the 10 Mb network) takes twice as long as for the local case would seem to suggest that distributed applications should be designed to minimize inter-machine communication. In general, one might consider it impractical to view interprocess communication as transparent across machines when the speed ratio is that large. However, a more realistic interpretation is to recognize that the remote operation adds a delay of about one millisecond, and that in many cases this time is insignificant relative to the time necessary to process a request in the server. Furthermore, the sending or client workstation processor is busy with the remote ***Send*** for only 1.30 milliseconds out of the total 2.23 millisecond time (using the 10 MHz processor). Thus, one can offload the processor on one machine by, for instance, moving a server process to another machine if its request processing generally requires more than 0.36 milliseconds of processor time, i.e. the difference between the local ***Send-Receive-Reply*** time and the local processor time for the remote operation.

The above observations relate primarily to the ***Send-Receive-Reply*** sequence. A number of similar arguments can be made for the ***MoveFrom*** and ***MoveTo*** primitives, although some additional considerations have to be taken into account. First, in order to establish a reasonable lower bound of the cost of a remote ***MoveFrom***, it is not quite accurate to take the sum of the local time plus the network penalty minus the amount of concurrent execution. The cost of a local copy has to be subtracted from this quantity in order to establish an accurate lower bound. Indeed, unlike for a ***Send*** operation, where a local copy of the message is made[6], no such copy is done for ***MoveTo*** and ***MoveFrom*** operations. On a 10 Mb network and a 10 Mhz

[6]The cost of this copy is small anyways, given the small size of the message.

processor, this argument leads to a lower bound of 5.06 milliseconds (The copy time is approximately 0.70 milliseconds.) This figure has to be compared with the experimentally measured time of 5.83 milliseconds. Relatively speaking, the experimental numbers for the ***MoveFrom*** operation are much closer to the lower bound than the numbers obtained for the ***Send-Receive-Reply*** sequence. This is explained by the fact that the ***MoveFrom*** operation does not require any dynamic buffering, since by its definition, buffers are available both in the process that is executing the ***MoveFrom*** as well as in the target process. Additionally, it can be observed that, relatively speaking, the amount of concurrent execution for ***MoveFrom*** operations is much lower than for the ***Send-Receive-Reply*** sequence (10 percent vs. 36 percent). This again indicates the dominant cost of the actual data transfer in the overall cost of the ***MoveFrom*** operation, a cost which cannot be decreased by concurrency, since its basic constituents, the sender copy into the network interface, the transmission and the receiver copy out of the network interface necessarily happen in series.

3.5.3. Comparison with Other Results

Comparison with other experimental results is always extremely difficult, given the subtle variations in the semantics of the primitives, and differences in hardware and measurement methods. The figures in this section are thus to be taken more as ballpark figures rather than as exact comparisons.

A V message exchange is very similar to Leblanc's *synchronous port call* [46]. Leblanc estimates that on a 1 MIPS processor and a 10 Mb network (similar to our prototype environment), a synchronous port call with 2 bytes of data would take 3.43 milliseconds. It is probably fair to assume that this cost includes a sizable constant term, independent of the amount of data in the call, and a second term, linear in the number of data bytes. Thus, one might expect synchronous port calls with 32 bytes, equal to the amount of data in V messages, to take slightly more than 3.43 milliseconds. This result is to be compared with 2.23 milliseconds for a V message exchange.

In his thesis, Spector reports that a remote memory reference of 16 bits between two 68000s connected by a 3 Mb network takes 152 microseconds, under favorable circumstances [71]. The amount of data transferred in a V message exchange would require 32 such remote references (32 bytes for both the ***Send*** and the ***Reply***), resulting in an overall cost of approximately 4.8 milliseconds.

Finally, a Cedar remote procedure call with 8 arguments and 8 results is reported to take 1.25 milliseconds between two Dorados on a 3 Mb Ethernet network [9]. About 0.25 milliseconds of that time is accounted for by network transmission, and thus approximately 1 millisecond is spent on the processors. Since a Dorado is an order of magnitude faster than a 68000, we might estimate approximately 10 milliseconds for a comparable operation on 68000. Of course, an 8 argument, 8 result Mesa procedure call entails substantially more functionality than a V message exchange.

3.5.4. Multi-Process Traffic

The discussion so far has focused on a single pair of processes communicating over the network. In reality, processes on several workstations would be using the network concurrently to communicate with other processes. Also, server processes would be accessed concurrently by many other processes. Some investigation is called for to determine how much message traffic the network can support, and what degradation in response time is to be expected as a result of other network or server load.

First, let us consider the effects of network load. A pair of workstations communicating via ***Send-Receive-Reply*** at maximum speed generates a load on the network of about 400,000 bits per second, about 13 percent of a 3 Mb Ethernet and 4 percent of a 10 Mb Ethernet. Measurements on the 10 Mb Ethernet indicate that for the packet size in question no significant network delays are to be expected for loads up to 25 percent [30]. Thus, one would expect minimal degradation with say two separate pairs of workstations communicating on the same network in this fashion. Unfortunately, our measurements of this scenario turned up a hardware problem in our 3 Mb Ethernet interface, which causes many collisions to go undetected and show up as corrupted packets. The response time for the 8 MHz processor workstation in this case is 3.4 milliseconds. The increase in time from 3.18 milliseconds is accounted for almost entirely from the

timeouts and retransmissions arising from this hardware problem (roughly one retransmission per 2000 packets). With corrected network interfaces, we estimate that the network can support any reasonable level of message communication without significant performance degradation. Similar measurements could not be conducted on the 10 Mb network due to the limited number of connected machines at the time of writing.

A more critical resource is processor time. This is especially true for machines such as servers that tend to be the focus of a significant amount of message traffic. For instance, just based on server processor time, a workstation is limited to at most about 575 message exchanges per second, independent of the number of clients. The number is substantially lower for file access operations, particularly when a realistic figure for file server processing is included. File access measurements are examined in the next section.

3.6. File Access Using the V Kernel

Although it is attractive to consider the kernel as simply providing message communication, the predominant use of message communication is to provide file access, especially in our environment of diskless personal workstations. File access takes place in several different forms: random file page access, sequential file access and program loading. In this chapter we restrict ourselves to random page-level access and program loading. Sequential file access is studied in detail in the next chapter. We assume that the file server is dedicated to serving the client process we are measuring and otherwise idle. We first describe the performance of random page-level file access.

3.6.1. Random Page-Level File Access

Tables 3-9 and 3-10 list the times for reading or writing a 512 byte block between two processes using the 10 MHz processor, interconnected by a 3 Mb or a 10 Mb Ethernet[7] . The columns are to be interpreted according to the explanation given for similarly labeled columns of Tables 3-5 to 3-8. The times do not include the time to fetch the data from disk but indicate expected performance when data is buffered in memory. A page read involves the sequence of kernel operations: ***Send-Receive-Reply***, whereby the page is appended to the ***Reply***. A page write involves a ***Send-Receive-Reply***, where again the page is appended, this time to the ***Send***.

There are several considerations that compensate for the cost of remote operations being higher than local operations (Some are special cases of those described for simple message exchanges.) First, the extra 2.9 millisecond time for remote operations is relatively small compared to the time cost of the file system operation itself. In particular, disk access time can be estimated at 20 milliseconds (assuming minimal seeking) and file system processor time at 2.5 milliseconds[8]. This gives a local file read time of 24.2 milliseconds and a remote time of 27 milliseconds, making the cost of the remote operation only 12 percent more than the local operation.

Random Page-Level Access

Kernel Operation	Elapsed Time	Network Penalty	Processor Time

[7] From this point, we present only the figures for the 10 Mhz processor.

[8] This is based on measurements of LOCUS [59] that give 6.2 and 4.3 milliseconds as processor time costs for 512-byte file read and write operations respectively on a PDP 11/45, which is roughly half the speed of the 10 MHz Motorola 68000 processor used in the SUN workstation.

Operation	Local	Remote	Diff.		Client	Server
page read	1.69	5.56	3.87	3.89	2.50	3.28
page write	1.68	5.60	3.94	3.89	2.58	3.32

Table 3-9: Page-Level File Access: 512 byte pages (times in msec.)

page read	1.69	4.54	2.85	3.15	2.41	3.08
page write	1.68	4.54	2.86	3.15	2.38	3.10

Table 3-10: Page-Level File Access: 512 byte pages (times in msec.)

This comparison assumes that a local file system workstation is the same speed as a dedicated file server. In reality, a shared file server is often faster because of the faster disks and more memory for disk caching that come with economy of scale. If the average disk access time for a file server is 2.8 milliseconds less than the average local disk access time (or better), there is no time penalty (and possibly some advantage) for remote file operations.

Second, remote file access offloads the workstation processor if the file system processing overhead per request is greater the difference between the client processor time for remote page access and for local page access, namely 0.7 milliseconds. A processor cost of more than 0.7 milliseconds per request can be expected from the estimation made earlier using LOCUS figures.

These measurements indicate the performance when file reading and writing use explicit segment specification in the message and the kernel appends the segments appropriately. However, a file write can also be performed in a more basic Thoth-like way using the ***Send-Receive-MoveFrom-Reply*** sequence. For a 512 byte write on a 10 Mb network, this costs 7.0 milliseconds. Thus, the segment mechanism saves 2.5 milliseconds on every page read and write operation, justifying this extension to the message primitives. A caveat needs to be made about the benefit of being able to receive the segment at the same time as the message is being received. In the current implementation the segment gets dropped when the receiver is not waiting to receive a message at the point the network packet arrives. This is more likely to happen under conditions of increasing load, and could thus deteriorate the performance to the figure mentioned above for the ***Send-Receive-MoveFrom-Reply*** sequence. When a demand paging becomes available, we hope to avoid this problem by always accepting the incoming segment and mapping it into the receiver's address space at the time the receiver performs the next ***Receive***.

3.6.2. Program Loading

Program loading differs as a file access activity from page-level access in that the entire file containing the program (or most of it) is to be transferred as quickly as possible into a waiting program execution space. For instance, a simple command interpreter we have written to run with the V kernel loads programs in two read operations: the first read accesses the program header information; the second read copies the program code and data into the newly created program space. The time for the first read is just the single block remote read time given earlier. The second read, generally consisting of several tens of disk pages, uses ***MoveTo*** to transfer the data. Because ***MoveTo*** requires that the data be stored contiguously in memory, it is often convenient to implement a large read as multiple ***MoveTo*** operations. For instance, our current VAX file server breaks large read and write operations into ***MoveTo*** and ***MoveFrom*** operations of at most 4 kilobytes at a time. Tables 3-11 and 3-12 give the time to transfer 64 kilobytes between processes (The elapsed time for file writing is basically the same as for reading and has been omitted for the sake of brevity.) The transfer unit is the amount of data transferred per ***MoveTo*** operation in satisfying the read request.

The times given for program loading on the 3 Mb Ethernet using a 16 or 64 kilobyte transfer unit corresponds to a data rate of about 192 kilobytes per second, which is within 12 percent of the data rate we can achieve on a SUN workstation by simply writing packets to the network interface as rapidly as possible. Moreover, if the file server retained copies of frequently used programs in memory, much as many current timesharing systems do, program loading could achieve the performance given in the table, independent of

disk speed. Thus, we argue that ***MoveTo*** and ***MoveFrom*** with large transfer units provide an efficient program loading mechanism that is almost as fast as can be achieved with the given hardware.

Program Loading -- 3 Mb

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
Transfer unit	Local	Remote	Difference		Client	Server
1 Kb	71.7	518.3	446.5	434.5	207.1	297.9
4 Kb	62.5	368.4	305.8	*[9]	176.1	225.2
16 Kb	60.2	344.6	284.3	*	170.0	216.9
64 Kb	59.7	335.4	275.1	*	168.1	212.7

Table 3-11: 64 Kilobyte Remote Read (times in msec.)

[9] Not available from measurements.

Program Loading -- 10 Mb

Kernel Operation	Elapsed Time			Network Penalty	Processor Time	
Transfer unit	Local	Remote	Difference		Client	Server
1 Kb	71.7	376.1	304.4	328.4	175.8	246.6
4 Kb	62.5	241.1	178.6	*	147.8	181.4
16 Kb	60.2	206.7	146.5	*	140.6	163.9
64 Kb	59.7	198.0	138.3	*	138.9	159.5

Table 3-12: 64 Kilobyte Remote Read (times in msec.)

3.7. Pipes

3.7.1. Introduction

One of the principal claims of this thesis is the utility of a general interprocess communication mechanism as a communication substrate for a distributed system. The main advantage of interprocess communication mechanisms is their generality, allowing other protocols to be built on top in a convenient fashion. With special-purpose protocols, a new protocol has to be devised from the ground up for every new application. However, if proper care is not taken, the extra layer of protocol can cause significant performance degradation.

In Section 3.6 we refuted the notion that the use of interprocess communication as a base for file access results in unsatisfactory file access performance. We continue the discussion of file-access performance in Chapter 4. In order to fully substantiate our claim about the usefulness of interprocess communication, we still need to demonstrate that other protocols (other than file access) can be built conveniently and efficiently on top of our interprocess communication mechanism. We have chosen to illustrate this point by describing the implementation and the performance of *pipe* data transfer, when implemented on top of the V kernel primitives. From a communication standpoint, pipes are different from file access in that there exists a *symmetrical* relationship between the two communicating entities: the reader and the writer of a pipe have similar responsibilities and privileges. Between a user program and a file server, an *asymmetrical, client-server* relationship exists: the client sends requests and the server fulfills them. A pipe on the other hand is a form of client-to-client communication. The alert reader undoubtedly has noted that the asymmetric nature of the V message primitives, with distinct primitives for clients (***Send***) and servers (***Receive***, ***Reply*** and ***Forward***), models very well the client-server style interaction between a file system and its clients. Client-to-client communication cannot be accomplished as directly with the V primitives: clients typically only execute ***Sends*** and never execute ***Receives***[10]. One could go from there and argue that we have sufficiently "adjusted" our message primitives to the desiderata of file access, and in doing so, we have undone the benefits we hoped to achieve by using interprocess communication, namely that all applications could be on top of the interprocess communication layer, in a convenient and efficient fashion. In this section we address this objection by presenting the implementation and the performance of pipes in the V-System. In particular, we show that the experimentally measured maximum data rate for V network pipes is similar to the estimated maximum data rate of a kernel-level implementation. For local pipes, some performance loss has to be tolerated, although small on an absolute scale.

Pipes are described as follows. In Section 3.7.2 we describe the semantics we expect of pipes. In Section

[10] Note that this problem is not specific to the V communication primitives, but is present in any interprocess communication system that espouses a client-server model. With remote procedure calls, for instance, servers provide procedure bodies and clients make procedure calls. There is no way for direct client-to-client communication because clients do not provide procedure bodies for invocation by other clients.

3.7.3 we describe the implementation of pipes in terms of the V primitives. The performance of pipes in this implementation is presented in Section 3.7.4. Next, in Section 3.7.5, we estimate the potential performance of pipes in a kernel implementation, with a specialized protocol for network pipes. In Section 3.7.6 we take a look at related efforts in the LOCUS and Eden systems. Conclusions on the subject of pipes are drawn in Section 3.7.7.

3.7.2. Pipe Semantics

A *pipe* is a communications paradigm that provides for the buffered and synchronized transmission of streams of data between processes. Next, we list the requirements a pipe implementation should satisfy.

Operations — A process can *read* data from a pipe or *write* data to a pipe in blocks of a fixed maximum size. Additionally, there are operations for creating, opening and closing pipes.

Buffering — A pipe should provide some amount of buffering between reader and writer. The buffered data should continue to be accessible to the reader after the writer has gone away.

Synchronization — A process trying to read from a pipe that is "empty" (i.e. has currently no data blocks queued for reading) is suspended until more blocks become available for reading. Trying to write to a pipe that has used up all of its buffers causes the writer to block until some buffers are freed up by reads.

Pipe end movement — It should be possible to change the reading and/or writing end of a pipe. It should also be possible to have multiple readers and writers to a single pipe. It is assumed that concurrent access is serialized in some convenient fashion, for instance by a token passing mechanism.

Maintaining ordering — The order in which blocks are written should be preserved on the reading end, even if more than one process (on more than one machine) writes to the pipe.

Exception processing — An end-of-file indication is returned to a process trying to read from an empty pipe whose writing end has disappeared. A write to a pipe whose reading end has gone away, is reflected as successful to the writer. The data of such a write is discarded.

V pipes are very similar to Unix pipes [65], and a natural extension of the latter in a multi-machine environment. Unlike in Unix, a writer is not explicitly notified (by a ***sigpipe*** signal) when the reader of a pipe disappears.

3.7.3. Implementation of Pipes Using V Messages

In a system based on specialized protocols pipes are likely to be implemented as a separate protocol in the kernel. In the V-System, pipes are implemented outside of the kernel, by a process called the *pipe server*. This process uses the V communication primitives to communicate with its clients. Since the pipe server is accessed through the V message primitives, its location is transparent to its clients (except maybe for performance).

Once a pipe connection between two processes is established, the writer can execute a write on a pipe by sending a message to the pipe server containing a write request. Since the maximum data block size (1024 bytes) allowed for a single pipe write fits within the size of a segment that can be appended to a message, both the message and the data block are transmitted in a single operation (a single packet in the case of network access). If the pipe server is waiting to receive, it receives the message and moves the data block into its *current buffer* using the ***Receive*** operation. It then links the buffer into the list of buffers written but not yet read for this pipe. If, however, the pipe server is not waiting for a message at the time the write request arrives, the kernel queues the message but discards the data segment. After receiving the message, the pipe

server then uses the ***MoveFrom*** operation to get the data block. Having moved the block into its buffers, by either of the above methods, the pipe server then normally sends a ***Reply*** to the writer, except when the pipe has used up all the buffers it had been allocated. In this case, the ***Reply*** is delayed until a buffer is freed by a read request. Thus, blocking the writer when the pipe is full is done by delaying the ***Reply*** to its last write request.

Similarly, reading from a pipe is accomplished by sending a message with a read request to the pipe server. If buffers are queued for this pipe, the server unlinks the first buffer out of the queue and transmits it to the reader. Again, since the maximum pipe data block fits within the maximum size for appending segments to messages, this can be done in a single operation by appending the block to the ***Reply*** message. If no buffers are available to be read, the reader is blocked by delaying the ***Reply*** until new buffers are written.

So, under normal circumstances, two message exchanges (one for the read and one for the write) are necessary to get a block from the reader to the writer. If the pipe server is not ready to receive when a write request comes in, an extra ***MoveFrom*** is necessary. If all processes involved, the reader, the writer and the pipe server are on different machines, this leads to 4 packets in the former case, and 6 in the latter (assuming no retransmissions). If the reader is located on the same machine as the pipe server, with the writer on a different machine 2 (4) packets are necessary. Finally, when the writer and the pipe server are colocated, with the reader on a different machine, the exchange should require 2 packets in either case (See Figure 3-9).

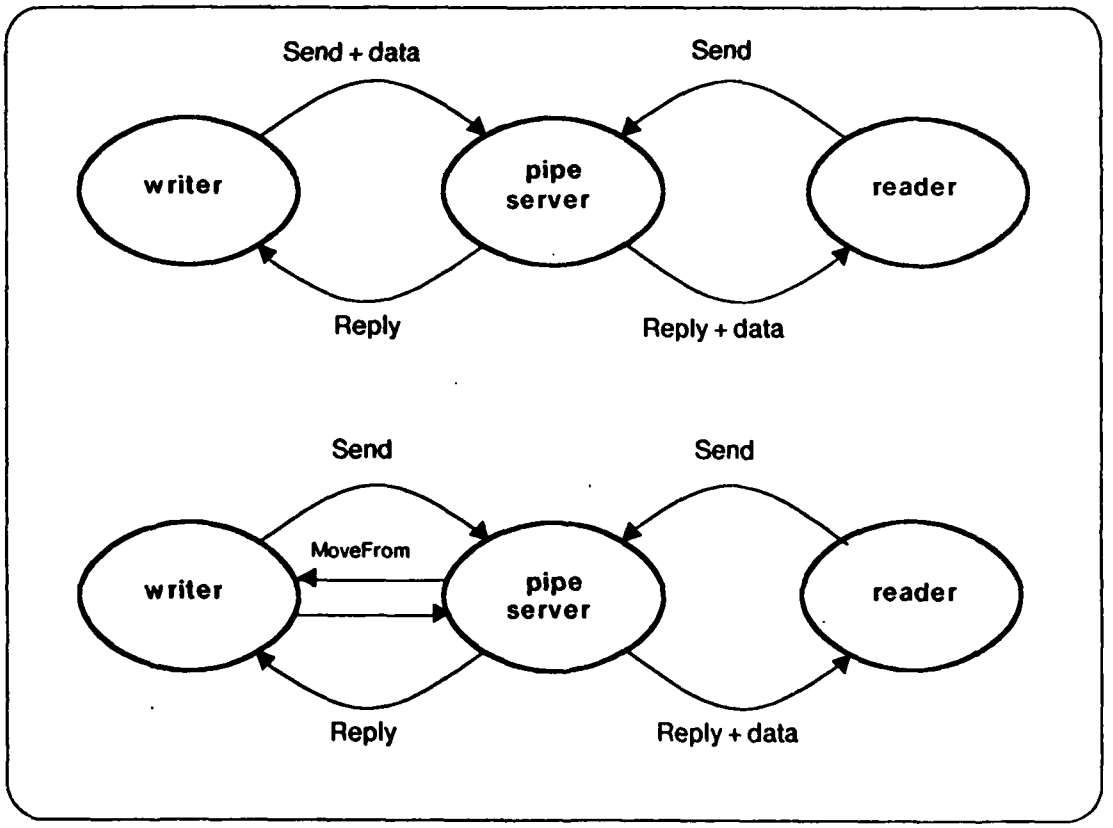


Figure 3-9: Pipe Data Transfer: Message Traffic

3.7.4. Performance of V Pipes

In this section we present the performance figures obtained experimentally for V pipes. Table 3-13 shows measurement results for V pipes on 10Mhz SUN workstations connected by a 3 Mb Ethernet. In the table we use the following notation: R for the reader process, W for the writer process and P for the pipe server. A single dash separating two letters indicates the processes corresponding to the letters are on the same machine; triple dashes indicate the two processes are on different machines. At the time of the measurements, few other network traffic was present, so network contention should be all but absent from our figures. Furthermore, the pipe server, the reader and the writer process were the only processes running on the respective machines. All transfers were done at the maximum pipe block size of 1024 bytes. We now analyze how the total cost of transferring a block through a pipe breaks down over the different components in its implementation. The primary cost components are the two ***Send-Receive-Reply*** sequences, with the appropriate data appended plus the overhead in the pipe server. In the local case, the two ***Send-Receive-Reply*** sequences add up to 4.25 milliseconds. The remaining 1.25 milliseconds must be accounted for by buffering and other pipe server overhead. One of the design decisions made in the implementation of the pipe server was to allocate data buffers dynamically rather than statically, in order to reduce the static code size of the pipe server. Reversing this decision could probably reduce the buffering overhead by some amount. However, it would increase the static code size of the pipe server, making it less attractive to station a pipe server on every machine.

V Pipe Performance

Configuration	Elapsed Time	Data Rate
R - P - W	5.5	180
R - P --- W	22.0	45
R --- P - W	10.1	100
R --- P --- W	18.2	55

Table 3-13: V Pipe Performance (times in msec.,data rate in Kbytes per sec.)

When the reader and the writer are on different machines, and the pipe server is located with the writer, the elapsed time is 10.1 milliseconds. This data transfer is made up of a local message exchange with 1 kilobyte of data (2.13 milliseconds), a remote message exchange with 1 kilobyte of data (9.5 milliseconds), and the pipe server overhead (again estimated at 1.25 milliseconds). This adds up to 12.8 milliseconds, indicating some amount of concurrency between the two machines (The two machines are concurrently active during about 27 percent of the time it takes to do do the transfer.)

When both the reader and the writer are on a different machine from the pipe server, performance drops approximately by a factor of 2 vs. the case of the writer and the pipe server being on the same machine. This is explained by the fact that now two remote message exchanges (with the appropriate segments appended) have to transpire. Using the same argument as in the previous paragraph, this would amount to 20.2 milliseconds vs. an empirically observed value of 18 milliseconds, again indicating some amount of concurrency.

The performance is comparatively poor in the case of the writer being on a different machine from the reader and the server. At first glance, there is no reason to expect the performance of this case to be different from the performance measured for the case of the reader being on a different machine from the writer and the pipe server. The poor performance is partially due to an artifact of the measurement and partially to the current implementation of ***Receive***. Remember that the segment appended to a ***Send*** gets discarded when the receiver (in this case the pipe server) is not waiting to receive a message. In this case, the sequence of events is as follows. The pipe server does a ***Receive***. The first message from the writer arrives with the data block appended to it. Since the pipe server was waiting for a message, the segment gets read in immediately. The write request is replied to and the pipe server proceeds to transmit the packet to the reader using ***Reply***. However, in the mean time, the writer has received the ***Reply*** to its first write and sends off its second request.

Due to the particular timing of events in this experiment, this message arrives at the server before the latter is done with the ***Reply*** and therefore the segment gets discarded. Now when the server gets ready to receive the message, it has to do a ***MoveFrom*** before it gets the data block. This sequence of events repeats itself for every block and causes performance inferior by about a factor of 2 to what one would be lead to expect. In practice, one would expect that some amount of processing between blocks by both the reader and the writer would prevent this peculiar sequence of events from happening.

These results suggest the following strategy:

1. Have a pipe server resident on every machine.

2. Have a particular pipe managed by the pipe server on the machine on which the writer resides.

The first part of this strategy is easily accomplished. Since the static code size of the pipe server is quite small (11 Kbytes), this does not impose any heavy memory demands on the workstation. The pipe server can easily be loaded as part of the system processes that are loaded when booting the workstation. The second part presents somewhat of a problem and is not fully implemented right now. Typically, a pipe is created by an executive, and then both ends of the pipe are passed on to processes created by the executive. Currently, the executive always asks the pipe server on its machine to create the pipe and this pipe server manages the pipe throughout its existence. If the writing end is passed to a process on a different machine, performance is suboptimal. This strategy is adequate for our current environment. However, once more sophisticated remote execution facilities become available, we might review this decision. This issue is a special case of the more general problem of efficiently supporting multiple writers to a pipe on different machines. We consider this problem next.

Consider the case of multiple writers to a pipe, some of which reside on different machines. In our current implementation, all data passes through a single pipe server, located on the machine where the pipe was created. At first glance, it would seem that this approach results, quite wastefully, in an extra "hop" for all data not written from the pipe server's machine. It would seem advantageous to move the data directly from the writer's machine to the reader's machine, without intermediate. However, the situation is severely complicated by the requirement (See Section 3.7.2) that the order in which blocks are written to the pipe is preserved at the reading end. For instance, if the creator writes something into the pipe, then passes on the writing end to a process on another machine, and then this process writes into the pipe, the order of writing has to be preserved on the reading end. A single, centralized pipe server automatically accomplishes this preservation of order. If multiple pipe servers are buffering data for a single pipe (with the objective of having the pipe server on the same machine as where the data is produced), there must be some additional protocol to ensure preservation of order. A possible approach would be to have a token passing mechanism, whereby only the holder of the token would be allowed to write into the pipe. Before the token can be passed and a new process is allowed to write into the pipe, the current pipe server must be informed that there will be a new writer and the new pipe server must be determined. The current pipe server could either forward the buffers it has queued for reading to the next pipe server, or it can keep the buffers but provide the reader with the address of the new pipe server when all of its buffers have been read. If moves are relatively infrequent, such a strategy results in better performance than the centralized pipe server. At present, however, we have deemed this complication unnecessary. Note that buffering the data in the pipe server on the reading site (even ignoring the performance difficulties discussed earlier with this approach) does not solve the problem because of the possibility of the reading end being passed to a different machine.

3.7.5. A Kernel-Level Implementation

In this section we provide a reasonable estimate of the maximum data rate that could be accomplished through a kernel-level implementation of pipes. The nature of the buffered communication mode associated with pipes requires that at least two copies of the data be made: one from the writer to the intermediate buffer and one from the buffer to the reader. Either one or both of these copies could be subsumed by unmapping the virtual memory page from its original location and then mapping it into its final location. Alternatively, one can imagine ridding oneself of one of the copies by allowing the writer to fill a buffer, specify its address

and size to the kernel, and then allowing it to proceed without a copy of the buffer being made[11]. When the reader then asks for this block of data, it can be copied from its original buffer to the reader's buffer without an intermediate copy. This technique imposes on the writer the responsibility not to touch the corresponding buffer until notification has been received that the reader has made its copy of the data. Part of this process can be automated, again by using a virtual memory technique, called *copy-on-write*, whereby the kernel makes a copy of the relevant virtual memory page(s) only when the writer attempts to write into the buffer [63]. In order to be able to accurately compare a process-level and a kernel-level implementation, we assume that no page map trickery is used and that a copy is effectively made (Note that the implementation of the message primitives does not rely on such trickery either.) Besides the cost of copying data, a number of other factors have to be taken into account in order to come up with a reasonable estimate on the time it takes to transfer a block of data through a pipe. These include the overhead of the system calls to execute both the read and the write operations, process switching overhead and the cost of dynamically allocating buffers.

In the case of a pipe server-based implementation, the intermediate buffers reside in the pipe server. Data is moved from the writer to the pipe server and from the pipe server to the reader by means of the interprocess communication primitives. In a kernel-based implementation, the intermediate buffers reside in the kernel(s). If the reader and the writer are on the same machine, a copy is made from the writer into the kernel and then from the kernel to the reader. If the reader and the writer are on different machines, and the buffers are kept in one or both of the kernels on those machines, a copy is made from the writer to its kernel, from the writer's kernel to the reader's kernel and then from the reader's kernel to the reader. If the buffers are in a different kernel, for instance because the writer has moved and the buffers remain at their initial location, an extra copy is required into the intermediate kernel. In order to estimate the performance of a kernel-based implementation, we first estimate the cost of a process-to-kernel copy and a kernel-to-kernel copy. We estimate the cost of a process-to-kernel copy by the cost of an (intra-address) ***MoveTo*** operation of the same size. Indeed, the local ***MoveTo*** operation provides a means for moving data at a cost only slightly higher than the "raw" copy cost (See [15] for a further discussion). The fact that the cost is slightly higher stems from the overhead for access rights checking and for executing the ***MoveTo*** system call. Normally, one would expect the ***Read*** or ***Write*** system calls in a kernel-based implementation of pipes to incur a similar overhead. Thus, this estimate is quite accurate. The cost of a kernel-to-kernel copy is estimated by the cost of a remote ***MoveTo***: we have shown that remote ***MoveTo*** operations are a means for transferring data across the networks at speeds close to the network penalty, which is the best a kernel-to-kernel copy could achieve (Compare for instance the network penalty column to the elapsed time for a ***MoveTo*** in Tables 3-5 to 3-8.) While there is a slight additional cost for executing the system call, this overhead is negligible compared to the network penalty for the size of transfers we are considering (1 Kbytes).

Based on this argument, we estimate the cost for transferring a 1024 byte block through a local pipe to be the sum of the following components:

copy from the writer to the kernel	1.2 msec.
copy from the kernel to the reader	1.2 msec.
buffering overhead in the kernel	1.2 msec.
process switch between reader and writer	0.2 msec.

The buffering overhead in the kernel is assumed to be identical to the buffering overhead in the V pipe server. The process switching time is that observed for the V-System. The resulting cost is thus 3.8 milliseconds to be compared to the empirically measured 5.5 milliseconds for V pipes. In V, a pipe data transfer is performed by executing two ***Send-Receive-Reply*** sequences, one from the writer to the pipe server, and one from the pipe server to the reader. This results in four copies rather than two: since the pipe server, the reader and the writer are all in different address spaces, a copy must be made from the writer to the kernel's copy segment; from there to the pipe server; from the pipe server to the kernel's copy segment; and from there to the reader. These two extra copies, together with the extra process switch result in an extra cost of 1.7 milliseconds. Although significant on a relative scale (30 percent), the absolute value of the difference between both

[11] This technique is used in Charlotte in order to reduce the cost of asynchronous message communication [28].

implementations is quite small. We expect that with a maximum data rate of 180 kilobytes per second we are able to satisfy the needs of most applications.

For the two-machine case (reader and writer on different machines), it is more difficult to make a reasonable estimate because of potential concurrency effects. Temporarily ignoring such effects, we simply sum the cost of the different components of a two-machine pipe transfer:

copy from the writer to the writer's kernel	1.2 msec.
copy from the writer's kernel to the reader's kernel	8.0 msec.
copy from the reader's kernel to the reader	1.2 msec.
buffering overhead in the writer's kernel	1.2 msec.
buffering overhead in the reader's kernel	1.2 msec.

This adds up to a total of 12.8 milliseconds. Assuming an identical amount of concurrency as empirically observed for V pipes (27 percent) would result in an estimate of 10.1 milliseconds, identical to the value experimentally measured for the V-System. This surprising result is the consequence of the following factors. In the V implementation, data is buffered in the pipe server and only there. In the kernel-based implementation, data is buffered in both kernels. As a result, the buffering overhead is incurred twice and an extra copy is needed on the reader's machine, whereas in the V-System, data is copied immediately from the network interface into the reader's address space. These disadvantages of a kernel-based implementation are compensated by extra overhead in the V implementation resulting from the fact that in the latter data has to be copied from the writer to the pipe server (rather than from the writer to the kernel), the context switch between the writer and the pipe server, and finally the fact that the reader has to explicitly ask for the data across the network, whereas in the kernel-based implementation data is assumed to migrate to the reader's machine automatically.

In the case of multi-machine pipes, higher data rates could be achieved if one were to allow larger block sizes (spanning potentially multiple packets), and if one were to stream these packets across the network without intervening acknowledgements. In V, the same streaming could be accomplished by using ***MoveTo*** and ***MoveFrom*** with large blocksizes, rather than appending segments to the ***Send*** and ***Reply*** messages.

Finally, we wish to consider again the case of multiple writers. A kernel-level implementation of pipes must face the same issue as a pipe server-based implementation, namely the requirement that the order of blocks is preserved. The approaches suggested at the end of Section 3.7.4 remain valid: either all data passes through a single kernel and gets serialized there, or some protocol is executed between the different kernels to guarantee the appropriate ordering. As a final point in this comparison, we wish to compare the experimental figures from the V-System for the three machine case to an estimate for a kernel-based implementation for that case. Again, we temporarily ignore possible concurrency effects and sum the cost of the different cost components:

copy from the writer to the writer's kernel	1.2 msec.
copy from the writer's kernel to the intermediate kernel	8.0 msec.
copy from the intermediate kernel to the reader's kernel	8.0 msec.
copy from the reader's kernel to the reader	1.2 msec.
buffering overhead in the writer's kernel	1.2 msec.
buffering overhead in the intermediate kernel	1.2 msec.
buffering overhead in the reader's kernel	1.2 msec.

This leads to a total of 22.0 milliseconds, and again assuming an identical amount of concurrency as that observed experimentally in the V-System, to an elapsed time of 19.8 milliseconds. This value is slightly higher than that observed for the V-System (18.2 milliseconds). The explanation of this result is similar to the argument held for the two machine case. The kernel implementation pays a penalty by incurring the buffering overhead three times instead of once and by having to make an extra copy on the reader's machine. Its advantage over the V implementation results from the reader not having to go across the network to ask for the data.

3.7.6. Pipe Implementations in Other Systems

In systems based on problem-oriented protocols, one would expect pipes to be implemented as a separate protocol in the kernel. This is the case, for instance, in the LOCUS operating system. As in V, three sites can potentially enter into piped data transfer between two processes: the writer site, the storage site and the reader site. The logical flow of data is from the writer site to the storage site and from there on to the reader site. In first order approximation, one can think of the storage site as the equivalent of the pipe server in V. However, LOCUS uses a different buffering strategy. Unlike in V, where data is buffered in the pipe server process (and only there), all of the LOCUS kernels involved potentially provide intermediate buffering. The kernel on each of the three (logical) sites maintains its own view of the pipe, in particular of the pipe's current size and the read and write pointer. For instance on the reading site, the reader process reads from the pipe until the buffers in its kernel are empty. At that point, it is put to sleep and the kernel on that machine contacts the kernel on the storage site for more data. The storage site then returns any available data, and the pipe size in the storage site and in the reading site get updated accordingly. Similarly, on the writing site the writer process proceeds until its buffers are full, at which point its kernel contacts the storage site kernel. Data may also migrate asynchronously from the writing site to the storage site, and from the storage site to the reading site. Special-case code in the kernel provides the obvious optimizations when two or more logical sites coincide with a single physical site. From the available documentation, it would appear that the storage site for a particular pipe always remains at its initial location and that no attempt is made to bypass the storage site if both the reader and the writer are on a machine different from the one functioning as the storage site.

In the Eden system, a pipe-like interconnection mechanism, called *transput*, is provided. Eden objects consists themselves of several concurrent processes. The Eden transput mechanism takes advantage of this by having the data written by the writer object and not yet read by the reader object buffered by a separate process within the writer object. Thus, when a write is done to a pipe, the data is copied from the process within the writing object that performed the write to the buffering process. When a read comes in later, that request is directed to the buffering process which tries to satisfy it out of its buffers. Transput is thus accomplished without any kernel support other than the normal Eden invocation primitives. The motivation for this particular approach is to reduce the number of invocations per block transferred from two to one plus a much cheaper intra-object procedure call. This is particularly important in Eden because the invocation mechanism is rather heavy-handed and correspondingly expensive.

This optimization is made at the expense of a number of important concessions on the semantics of pipes. First, it seems that data disappears as soon as the writer disappears, contrary to one of the goals we set forth in Section 3.7.2. Second, the fact that data is buffered within one of the communicating entities removes a level of indirection between the reader and the writer. This level of indirection is quite useful when the writer changes over the pipe's lifetime. In V for instance, when the writing end changes, this does not affect the reader: it just continues to send messages to the pipe server. In Eden, it is necessary for the reader to be connected to the writer directly. When the writer changes, this connection has to be explicitly changed, presumably by means of an additional primitive in the reader. The multiple write case is even more problematic. In this case, a pipe server-like Eden object must be interposed between the different writers and the reader.

As a final note on this subject, we wish to point out that the V-System allows a similar multi-process structure as the Eden system, with multiple processes forming a single *team* and residing in a single address space. The library function for ***Write*** could create a process that would perform the same function as the buffer process in the Eden writer object. For a local pipe transfer, the cost would then equal the sum of a local (intra-address space) ***MoveTo***, a local ***Send-Receive-Reply*** with the appropriate data appended, and the buffering overhead, resulting in a total of 4.6 milliseconds, or 16 percent better than the measured V performance. For the two machine case, the local ***Send-Receive-Reply*** has to be replaced in the calculation by a remote one, resulting in a total of 11.9 milliseconds. Assuming again an identical amount of concurrency as empirically observed, the elapsed time would be 9.4 milliseconds or 7 percent better than the measured V performance. We deem these gains insignificant compared to the concessions that must be made on the semantics of pipes in order to accomplish them.

3.7.7. Conclusion

In this section we have compared the experimentally measured performance of pipes in the V-System to the estimated performance of a kernel-based implementation. In V, pipes are implemented using the interprocess communication facilities. There are no special kernel primitives for pipes, nor is there a special protocol to support network pipes. In a kernel-based implementation, there are evidently read and write kernel primitives, plus a protocol to implement them across machines.

The goal of this section was to make an assessment of the cost of layering pipes on top of the V interprocess communication, in comparison to a kernel-based implementation. We have shown that the maximum data rate through V network pipes compares very well with the estimated maximum data rate of a kernel-based implementation, because extra buffering in the kernel-based implementation compensates for the additional overhead of the V message passing primitives. For local pipes, some penalty has to be paid because of extra copies necessary in the V-System. While percentage-wise the penalty might seem high (30 percent), the difference in absolute terms is rather small (1.7 milliseconds per kilobyte).

We have also discussed the problems associated with the ends of pipes moving away from their original machine. So far we have found the simple approach of a single, centralized pipe server satisfactory although this assessment might change as more sophisticated remote program execution facilities become available. We have shown that this case could be handled efficiently with appropriate modifications to the pipe server.

Finally, we have presented details of pipe implementations in different system. For the Eden system in particular, this has allowed us to demonstrate the cost of certain aspects of the definition of pipes. When the definition is somewhat relaxed as in Eden, certain performance benefits can be accomplished.

3.8. Chapter Summary

In this chapter, we have presented the V interprocess communication primitives and their performance. We have also shown the performance of two applications, files and pipes, which are from a communications viewpoint significantly different. By comparing the measured performance of these applications to the optimally achievable performance in the given hardware environment, we have shown that the penalty for implementing these applications on top of the V interprocess communication primitives is small and that therefore, the benefits to be expected from specialized protocols for these applications are minimal.

The measurements in this chapter are taken in an otherwise unloaded environment and reflect performance in a particular hardware environment. In the next chapter, we use these measurements as inputs to a queueing network model of network page-level file access. This analysis leads to an understanding of the performance in different hardware environments and under more reasonable load conditions.

— 4 —
A Queueing Network Model for Remote File Access

4.1. Introduction

In the previous chapter we have concentrated on measurements of V interprocess communication in a particular hardware environment and in an otherwise idle system. While interesting in their own right, these measurements convey in themselves little information as to how users would perceive the overall performance of the V-System in a more realistic environment. First, users perceive the efficiency of a system through the performance of the applications they run, and not through the cost of individual low-level communication primitives. In this chapter, we consider *sequential file access* as our application of interest, because of its foremost importance in determining overall system performance. We compare remote vs. local sequential file access, a topic which is of paramount importance in our environment of diskless workstations. Second, in a realistic environment, several (diskless) workstations are competing for the services of the file server. This introduces queueing delays at the network, the file server and the disk, which enter into the user's perception of the performance of the system. Third, we would like to predict how system performance changes as some design parameters are changed. Among the parameters we consider are both hardware-related measures such as network data rate, processor speed and disk characteristics as well as software-related parameters such as the size of the interaction between clients and the file server, and various buffering and caching schemes.

In order to address these questions, we build a queueing network model of the system under consideration. This allows us to predict performance under different loads as well as in different environments, where measurements would be difficult or impossible. Starting from measurements on the V-System, we first compare the performance of local vs. remote sequential file access when only a single client accesses the file server. These measurements are then used as inputs to a queueing network model, allowing us to predict the performance degradation of remote sequential file access under load. In order to assess the benefits of a particular modification to the baseline system, we extrapolate from the experimental measurements what the results would look like for the modified system. By feeding suitably modified inputs into the queueing network model, we can predict the performance improvements such a modification would yield.

The outline of this chapter is as follows. In Section 4.2 we present the canonical system under consideration and its representation as a queueing network model. Section 4.3 contains measurement results used as input data to the model and the evaluation of the model in this baseline configuration. Next, in Section 4.4, we examine the effects of various modifications to the baseline model. Finally, in Section 4.5, we summarize the basic conclusions from this modeling study. A similar study based on Unix 4.2BSD [52], with some comparison to V and Apollo DOMAIN [44], appears in [42].

4.2. The Canonical System

4.2.1. The System and its Model

The canonical system under consideration consists of a number of diskless workstations accessing a file server over a local area network (See Figure 4-1). The workstations use the file server for all permanent storage. We do not consider the effects of paging in this model. It is assumed that either the paging rates are very modest or that a local disk is available for paging.

Figure 4-2 shows the queueing network model used to represent the system. We use the terminology of [43]

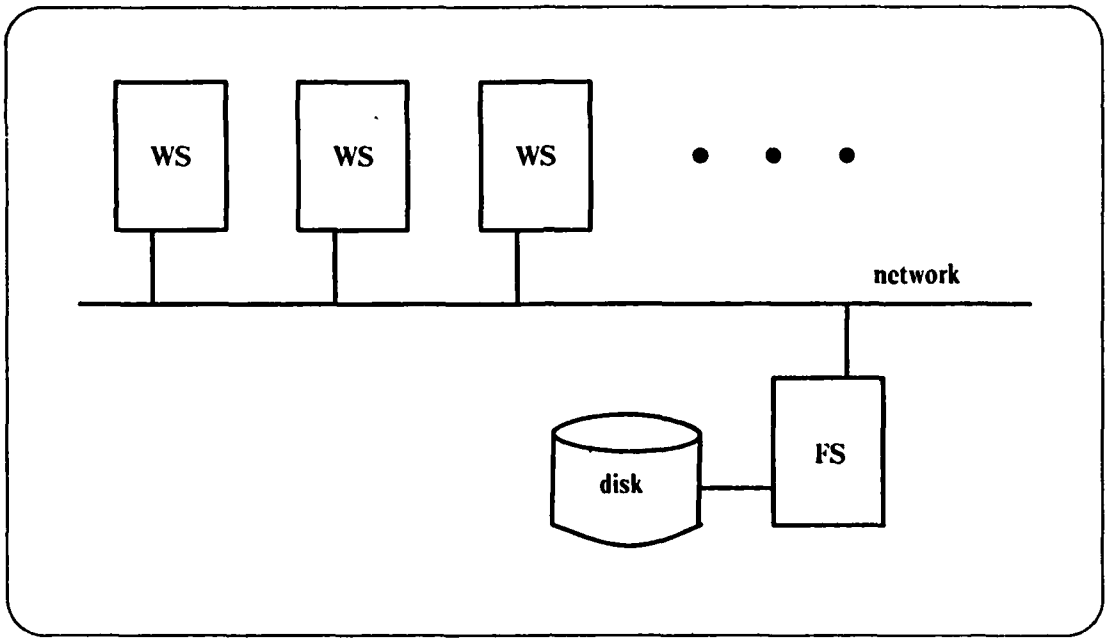


Figure 4-1: The Canonical System

to describe the different components. Resources are represented by *service centers*, with a specific *service demand* per request. Clients or *customers* offer a certain *workload* to the service centers. We assume that at any given moment there is a fixed number N of workstations, resulting in a *closed* queueing network model, in which clients generate requests separated in time by intervals of average length Z (the *think time*). The model includes one token per request (Workstations can only have a single request outstanding at any given point in time.) The token cycles through the network, accumulating service and encountering queueing delays as it visits the various service centers. We are interested in obtaining from the model the *response time*: this is the average round trip time of the token around the network, from the moment the corresponding request is initiated until its completion.

In the model the file server is represented by two service centers: the file server CPU and the disk. Each workstation is represented by a single service center, namely its CPU. We assume that there is no contention and therefore no queueing delay at the client workstation CPUs. The network completes the list of service centers, interconnected in the topology shown in Figure 4-2. In order to fully specify the model, we need to characterize the workload generated by customers and the service demands per request at the various service centers. This is discussed next.

4.2.2. Customer Characterization

First, we need to select a definition of a customer *request*. We take a system-oriented view of a request, namely we let a request correspond to doing 8 kilobytes of I/O[12]. Alternatively, we could have taken a more user-oriented definition, whereby a (user) command would be taken as the basic request unit. Both approaches are quite possible (See for instance [29] for an example of the first approach), and using our

[12] Note that we do not necessarily assume that I/O is done in units of 8 kilobytes.

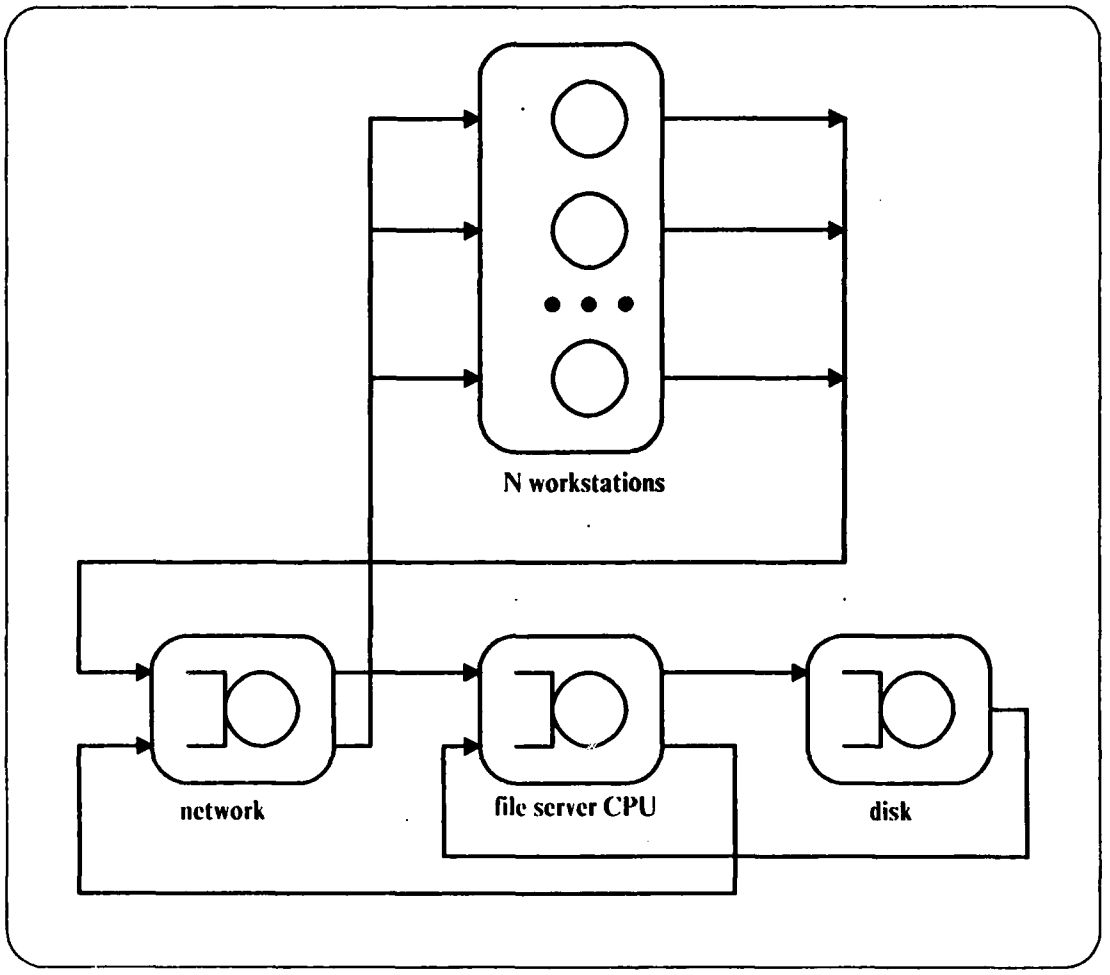


Figure 4-2: The Queueing Network Model Representing the Canonical System

solution techniques, both would yield the same result. We have somewhat arbitrarily chosen the system-oriented view because of the ease with which service demands can be measured for this type of request. The customer workload is then described by the average rate at which such requests are generated by a typical workstation user. Evidently, the request rate is strongly dependent on the kind of applications the system supports. We are interested in environments where users spend most of their time doing software development-related work, as in a typical Unix environment. Measurements done at the University of Washington on a comparable diskless workstation-based system indicate that workstation users generate on average approximately 8 kilobytes of I/O every two seconds [42]. Thus, we introduce a think time of 2 seconds in between customer requests.

4.2.3. Service Demands

Service demands per request have to be computed or measured for the network, the file server CPU, the disk and the client CPU. Additionally, we have measured the elapsed time for both local and remote operations. We work with the figures for reading only: the service demands for writing are not significantly different and the number of reads typically far exceeds the number of writes. The actual figures for the

baseline model are reported in Table 4-1 in Section 4.3. We briefly describe here the experiments performed and the assumptions made in arriving at these service demands.

The service demands are generated by conducting a set of experiments that repeatedly transfer 8 kilobytes of data between the client and the file server. Service demands (i.e. processor time) on the client workstation and on the file server are measured by running a low-priority "busywork" process that repeatedly updates a counter in an infinite loop (See Section 3.4). All other processor utilization reduces the processor allocation to this process. Thus, the processor time used per operation is the total elapsed time minus the processor time allocated to the "busywork" process divided by the number of operations executed. This method of measuring processor time accurately accounts for interrupt-level processing of network and disk I/O as well as processor degradation due to bus interference from DMA operations. Network service demand is computed by summing the length of the packets involved in the I/O operation and dividing that sum by the network data rate. Disk service demand has three components: transfer time, rotational latency and seek time. Transfer time and average rotational latency are taken from the manufacturer's device specifications. In order to compute the average seek time, we measure the average seek distance (in cylinders). The average seek time (in milliseconds) is then derived by taking the corresponding seek time from the manufacturer's specifications.

In order to compute the service demand for the disk as well as in order to measure the elapsed time for the overall disk, some assumption needs to be made about the number of seeks that are going to be performed relative to the number of I/Os. In the measured baseline case we assume that one seek is performed for every I/O operation. In practice, we expect this to be an overestimation of the number of seeks. The V file system is extent-based and goes to great length to ensure that logically contiguous file blocks are also physically contiguous on the disk. The benefits of this strategy are moderated by the fact that in the kind of systems we are interested in, a large number of files are rather small. A Unix-like hierarchical file system typically fosters the use of many small files. Measurements of file sizes weighted by frequency of access at the University of Washington indicate an average of approximately 11 kilobytes [42]. Similar experiments done by Ousterhout at Berkeley yielded similar results [40]. Therefore, even if the file system were able to allocate all logically contiguous blocks in a physically contiguous manner on the disk, then we might still expect to see a significant number of seeks.

4.2.4. Solution Technique

In order to solve the model, we use an approximate single-class MVA (mean-value analysis) solution technique [64, 67] (For a good description of the application of MVA to queueing network problems, see also [43].) This technique has the advantage of being relatively accurate as well as computationally inexpensive but it imposes some limitations on the class of systems that can be successfully modeled. Two characteristics of our system defy proper formulation in the MVA framework, namely the load-dependent service demand on the Ethernet and the asynchrony in typical file system access. We briefly describe the approximations we use in order to account for these characteristics.

Asynchrony cannot be modeled directly in the MVA framework because this solution technique requires that there be only a single token per request present in the network. This token travels around through the network accumulating service at the different service centers. Thus, no request can be processed at two service centers at the same time. If a given request is the only request present in the system, the response time is equal to the sum of its service demands at the various service centers. Measurements of file access, as presented in Section 4.3, indicate -- as expected -- the opposite: in Table 4-1 the sum of the service demands is higher than the elapsed time, indicating some amount of parallelism. This is the result of a number of factors, including concurrent execution on the workstation and the file server (as indicated by the parallelism for interprocess communication in Tables 3-5 to 3-8), and file server CPU processing while the disk I/O operation is in progress. Methods have been proposed for incorporating asynchrony into MVA solution methods [32]. We have chosen not to use these methods because they add an extra class to the model, resulting in a significant increase in the amount of computation necessary to arrive at a solution. Rather, we use the following simple approximation: we subtract the amount of concurrency measured in the single-user case from the predictions of the model under all load conditions. This is clearly correct for the single-user case and

approximately true at low loads. Furthermore, under high load conditions, the effect of concurrency is negligible compared to the delays introduced by queueing.

The load-dependent characteristic of the Ethernet network is accounted for by a technique called *hierarchical modeling* [43]. Hereby, a low-level analytic and simulation model is used to derive the service demand of the Ethernet in function of its offered load [4]. This low-level model is then integrated into the overall queueing model of the system as a flow-equivalent (load-dependent) service center. An extension to the approximate MVA solution method is used to take into account the load-dependent nature of this service center [80].

4.2.5. Response Time

Given the topology of the network (as in Figure 4-2), the service demands at the various service centers and the customer workload, the MVA solution technique generates (among other things) the average response time for a request. By running the model with suitably modified service demands, we get the response time for modified versions of the system. We argue that these figures by themselves do not form an adequate basis for comparing the user's perception of the performance of different file service configurations. The user's perception of system performance is based on the speed of his applications. Besides file I/O, applications contain some amount of computing as well. A more adequate basis for comparison is therefore the sum of the average I/O response time (for a given file service configuration) plus the average amount of "user mode" computing corresponding to the amount of data in the I/O request. For instance, if applications contain on average 10 percent I/O and 90 percent computing, a two-fold increase in the time necessary to perform an I/O operation would look quite detrimental when viewed in isolation. However, when viewed in the context of the overall application, only a 10 percent performance decrease would result. A number of experiments were run in a comparable environment to obtain an estimate of the average amount of computing relative to a given amount of I/O. They indicate an average amount of 212 milliseconds of user mode computing per 8 kilobytes of I/O [42].

4.3. Results from the Baseline Model

4.3.1. Service Demands and Elapsed Times

In Table 4-1 we present the results of measurements of service demands and elapsed times for doing 8 kilobytes of I/O in the baseline configuration. This configuration consists of a client running on a SUN workstation, a file server running on a SUN with a Fujitsu Eagle disk and an Xylogics disk controller, both of them connected to a 10 Mb Ethernet network.

Baseline Configuration

	Local Read	Remote Read
Client CPU	25.44	22.28
Network		7.24
Server CPU		40.02
Disk	31.45	31.45
Total	56.89	100.99
Elapsed	50.93	87.42

Table 4-1: Service Demands and Elapsed Times for 8 Kilobyte Transfers in the Baseline Configuration

4.3.2. Discussion

The numbers in Table 4-1 allow us to make the following comparisons between local and remote single-user I/O performance.

1. The ratio of total service demands for remote vs. local file access is 1.78. This indicates that there is a significant cost to accessing files over the network, when the service demands for file I/O are considered in isolation (without regard to parallelism or user mode computing). This is true even though we make a rather pessimistic assumption about the disk (one seek per I/O). A more optimistic assumption about the seek vs. I/O ratio would further inflate the ratio of remote vs. local service demands.

2. Concurrency is more outspoken in the remote case than when the file server is local. The ratio of remote vs. local elapsed times of 1.70 is therefore somewhat lower than the corresponding ratio of total service demands.

3. When an average amount of user mode computing of 212 milliseconds per I/O operation is added to the elapsed time for file access, the ratio between remote and local drops to a more acceptable value of 1.15.

As argued before, we believe the latter ratio is the correct figure of merit in comparing different file service configurations. From the previous arguments, we conclude that when the file server is unloaded, and the file server is identical to the workstation (with respect to CPU and disk speed), the performance penalty for accessing a remote file server is approximately 15 percent. In the next section, we consider the evolution of this penalty as more workstations are added to the file server.

4.3.3. Effects of Congestion

Figure 4-3 shows the effects of congestion as more workstations are added to the system. This graph has the average amount of user mode computing per I/O operation added to the elapsed time for file I/O. With 10 workstations the ratio of remote vs. local is 1.19, to be compared against 1.15 in the single-client case -- a very moderate increase. For 30 workstations the ratio has risen to 1.45 and starts to rise sharply thereafter.

The results for the baseline model under single-user and multi-user access so far allows us to make the following conclusion. Assume we are willing to pay some performance penalty for the benefits of a shared remote file server (lower cost per workstation, easier file sharing, etc.) Let us somewhat arbitrarily fix an upper limit on this penalty at 20 percent above local file access times. In that case, remote file access from a workstation to a shared file server -- under the assumptions of the baseline model -- has acceptable performance as long as the number of workstations is kept below 12, when viewed in the context of the average amount of user mode processing per I/O.

We now analyze the reasons behind the deterioration of the performance when more workstations are added to the file server. In order to do this, we use the following two intuitive results from queueing theory:

1. Under low load, response time is approximately equal to the sum of the service demands at the individual service centers. Therefore, a decrease of the service demand at any service center results in a corresponding decrease in response time.

2. Under high load, response time is primarily determined by the queueing delay in front of the service center with the highest service demand (the so called *bottleneck* service center). This queueing delay is in turn primarily determined by the service demand at that service center. Thus the most noticeable improvement in response time results from decreasing the service demand at the bottleneck service center.

These results are easily explained. First, under low load, a request cycles around the network, accumulating service at the service centers and only seldom encountering queueing delays. Clearly, the overall response time is approximately equal to the sum of the service demands. Second, as the load on the individual service centers increases, the queueing delays become more pronounced and at some point start to dominate the overall response time. Consider the (extreme) case where the bottleneck service center has a service demand

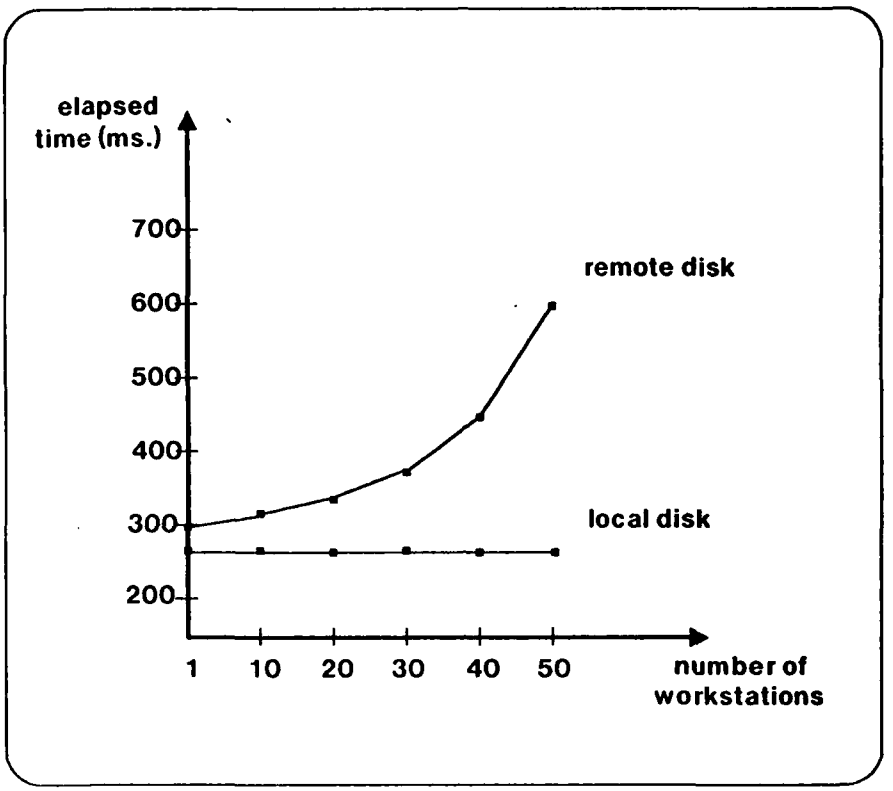


Figure 4-3: Multi-Client File Access: Response Time vs. Number of Clients

much higher than all the other service centers, and assume the load is so high that this center is entirely saturated (100 percent utilization). In this situation, requests spend almost all of their time in the queue waiting for service at the bottleneck service center. The length of this queue is approximately equal to N, the number of requests present in the system. As soon as a request has finished processing at the bottleneck service center, it quickly advances through the other service centers and almost immediately joins the queue the bottleneck service center again. The average amount of time spent in this queue (per cycle) is equal to the length of the queue seen on arrival (approximately equal to N) times the average service demand. Thus, since this queueing delay forms the biggest part of the overall response time, it can be seen that a decrease in service demand at the bottleneck service center results in an (approximately) N-fold decrease in overall response time. This result holds for the case where one service center has a service demand per request significantly higher than any other and under complete saturation of the network. Under less extreme assumptions, it remains true that a decrease in service demand at the slowest service center has the most beneficial effect on the response time under load: the effect is less than N-fold but still significant.

As Table 4-1 shows, the bottleneck service center in the baseline configuration is the file server CPU. Figure 4-4 confirms this observation: it graphs the utilization at the various service centers in function of the number of workstations. Clearly, the file server CPU approaches saturation as more workstations are added. Also noteworthy in Figure 4-4 is that the utilization of the network is truly minimal (less than 15 percent until 50 workstations are present).

More important than the exact shape of this graph, is the observation that the file server CPU rather than the disk is the service center with the highest service demand. This is true in spite of the fact that we made a pessimistic assumption about the seek vs. I/O ratio and in spite of the fact that the V interprocess

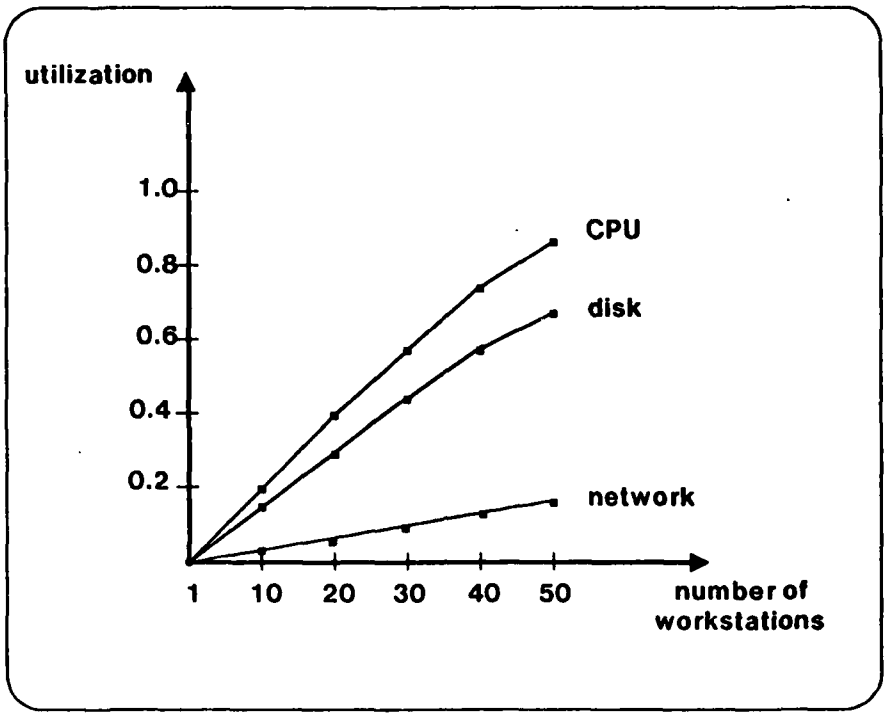


Figure 4-4: Multi-Client File Access: Utilization of Various Resources

communication is very efficient (Measurements of a Unix 4.2BSD system indicate a service demand on the file server that is twice as high [42].) Therefore, it is reasonable to expect that for a file server configuration with a CPU comparable to a 68000 and a disk comparable to the Fujitsu Eagle or faster, the file server CPU is the bottleneck service center. In the latter part of this chapter we are mainly interested in reducing the performance degradation under high load. Therefore, we investigate primarily those modifications that reduce the utilization of the file server CPU.

Finally, a point needs to be made about the validity of extrapolating the single-user measurements to multi-user access. In particular, some factors seem to indicate that the service demand per request on the file server CPU would be less under load conditions. This results from file server processes being continually active, rather than periodically, so that they do not need to be blocked and unblocked. This effect is difficult to quantify and somewhat dependent on the construction of the file server. Additionally, increased contention for file server buffers could potentially increase the service demand per request. We further ignore these effects and assume that the service demands are independent of the load on the system.

4.4. Modifications to the Baseline Model

In this section we are primarily interested in reducing the response time under conditions of high load. The different file server configurations we investigate are primarily aimed at this goal, although we also indicate their effect at low loads. Next is a list of the modifications considered in this section.

1. Using a faster file server processor.
2. Increasing the request size to the file server in order to reduce protocol overhead.
3. Disk caching on the file server.

4. Introducing some form of caching on the client workstation.

5. Increasing the number of file servers.

4.4.1. Faster File Server CPU

Figure 4-5 shows the effects of a CPU twice as fast as the 68000, as predicted by the model. These results are obtained by reducing the service demand on the file server CPU by half. The model shows significantly improved performance for all load values, but especially at high load. It is not clear that such a speedup could in practice be achieved. The protocol used to transfer large amounts of data (large with respect to the network packet size) is a streaming protocol without any form of flow control (See Chapter 5). Specifically, the source of the transfer sends out back-to-back maximum-size packets until all data has been transferred, without waiting for an acknowledgement between packets. In order to achieve high performance, the error rate has to be kept low. In particular, the destination (in this case the workstation) must be able to receive packets as fast as the source (the file server) can transmit them. Otherwise, the workstation starts dropping packets, causing the file server to retransmit, with overall performance degradation as a result. Clearly, when the speed of the file server is increased significantly, and it is sending out back-to-back packets, either the processor on the workstation has to be upgraded correspondingly or the network interface on the workstation must be capable of buffering a larger number of packets. Such modifications to the workstation might undo the economic arguments for diskless workstations. On the other hand, introducing flow control into the protocol most likely would (at least partially) undo the performance benefits one hopes to acquire by using a faster file server CPU.

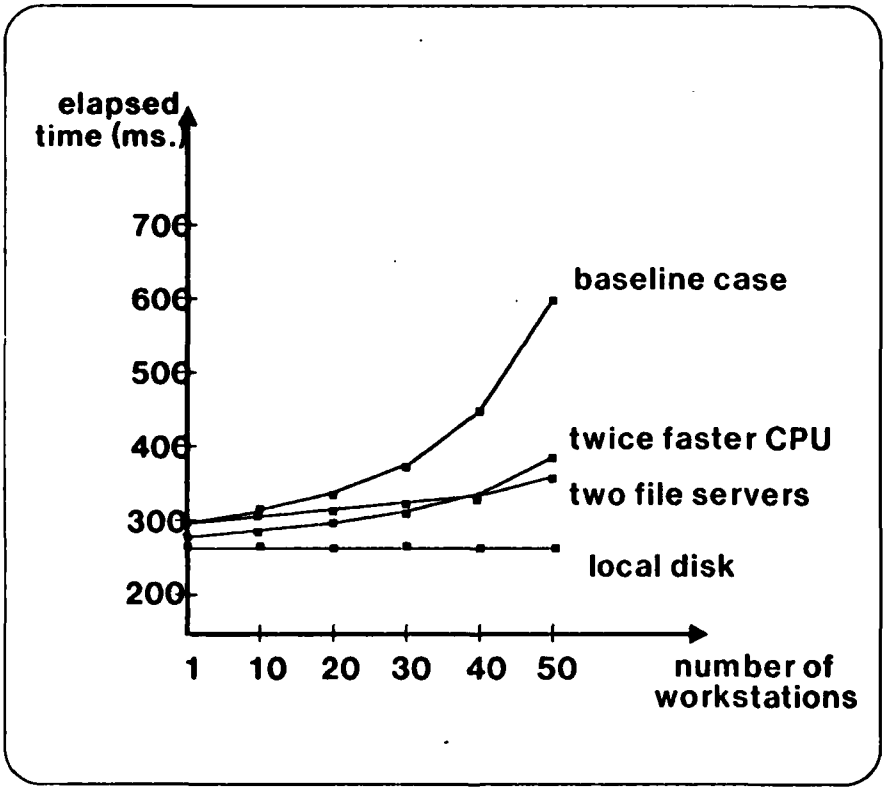


Figure 4-5: Remote File Access: Effects of a Faster File Server CPU

4.4.2. Increasing the Request Size

Increasing the size of client-server interaction decreases the demand on the file server by reducing protocol overhead as well as reducing the number of disk commands that need to be issued. However, similar observations as in Section 4.4.1 argue against increasing the request size much beyond 8 kilobytes. Again, in order to avoid buffer overflow and packet loss, it might be necessary to include more sophisticated network interfaces in the client machine. Additionally, while the benefits of going from 1 kilobyte requests to 8 kilobyte requests are very significant, the relative merits decrease as the request size is further increased. For instance, we measured an elapsed time of 410.80 milliseconds for reading 8 kilobytes using 1 kilobyte requests. This dropped to 100.99 milliseconds when using 8 kilobytes requests. A further increase of the request size to 64 kilobytes yielded only a modest improvement to 95.80 milliseconds. In terms of communication overhead, Table 3-8 indicates little benefit in going much beyond 8 kilobytes for ***MoveTo*** operations, both in terms of elapsed time as well as in terms of processor utilization. While in theory less seeking would be necessary, thereby reducing demands on the disk, in the kind of systems we are looking at, such a reduction of disk service demand would in practice not be accomplished. As mentioned before, in these systems most files are quite small, and therefore seeks would remain necessary.

4.4.3. File Server Caching

Figure 4-5 also shows the effect of caching the disk on the file server. The cache is assumed to have a 50 percent hit ratio. Such a cache reduces demand on the disk by avoiding to go to the disk for buffered pages. It also reduces service demand on the file server CPU by reducing the number of disk commands that have to be issued. On the other hand, cache management overhead increases the service demand on the CPU. The inputs for the model are generated as follows. Disk service demand is reduced by 50 percent. In order to obtain the file server CPU demand, an experiment is run whereby all requested pages reside in the cache. The processor service demand is measured for this experiment. The file server CPU demand used as input to the model is the sum of the service demand for the baseline case (including issuing the disk command) and the service demand for the above experiment, divided by two. Cache management overhead is ignored.

The model shows an improvement comparable to that obtained by using a file server twice as fast as the 68000. However, it does not pose the same problems as using a faster file server CPU. If cache management overhead can be kept low, this modification is very appealing. For instance, the file server could keep frequently used programs in memory, much as many contemporary timesharing systems do, thereby achieving a high cache hit ratio. At the time of writing, not enough data could be collected in order to judge whether a cache hit ratio of 50 percent is reasonable. Clearly, the hit ratio is strongly dependent on the workload present to the cache and the amount of memory that can be put to use as file server buffers.

4.4.4. Using a Client Cache

In addition to the advantages of a cache on the file server, a cache of file pages on a workstation machine reduces the load on the file server by eliminating the need for remote file access for pages that are found in the cache. On the down side, maintaining a cache on the workstation introduces cache management and cache consistency overhead. Part of this overhead is accounted for on the workstation, which is assumed not to be a source of contention, therefore its effects under load are minor. The caching overhead remains visible though in the low-load elapsed times. Some extra overhead would most likely also be incurred on the file server. Figure 4-6 shows the improvement in response time for a 50 percent cache hit ratio. The inputs for this model were half the measured file server CPU, disk and network service demand of the baseline model and identical client CPU service demand. Again, it is not possible to judge from currently available data whether the assumption of 50 percent cache hit ratio is reasonable. Presumably, a workstation cache could also be combined with a file server cache.

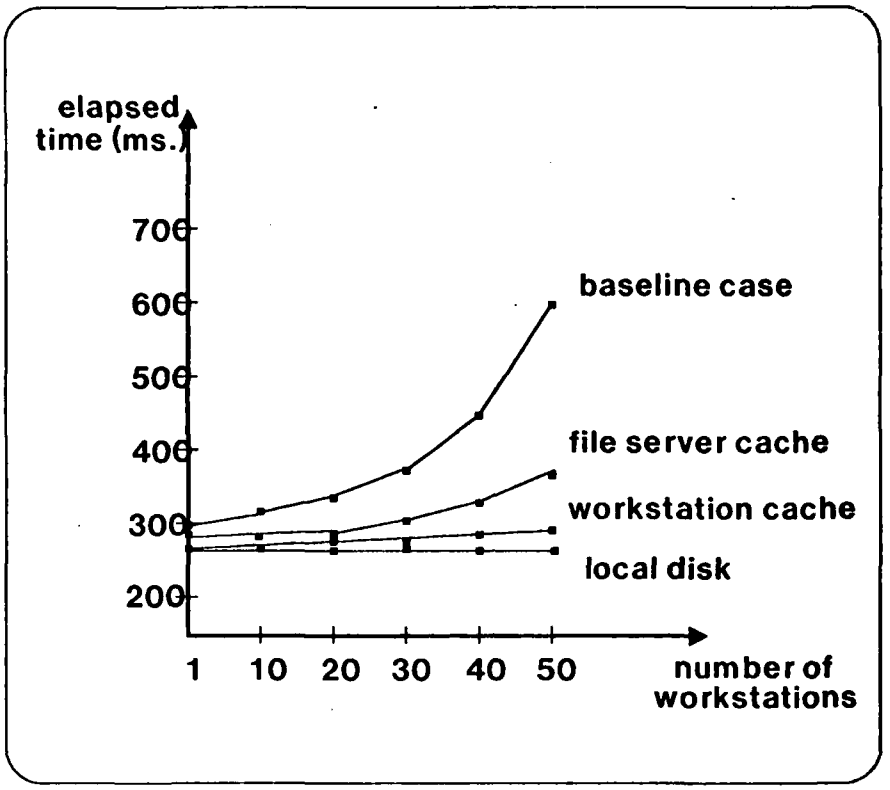


Figure 4-6: Remote File Access: Relieving File Server Load

4.4.5. Adding a Second File Server

In the previous section, we explored one way of decentralizing file access, namely by introducing caches of file pages on the workstations. In this section, we investigate an alternative way of decentralizing file access by considering the case of two file server machines. We assume that traffic is equally divided between the two file servers. Client CPU and network service demand thus remain the same, while for each file server, the disk and file server CPU demand of the baseline model are divided by two. Figure 4-6 shows the performance of this configuration. Performance improvement is comparable to that achieved by introducing a client cache with a 50 percent hit ratio. In practice, introducing a second file server would probably result in better performance than client caches since we ignored cache consistency overhead, if an equal spread of traffic can be achieved.

4.5. Chapter Summary

We have constructed a queueing network model of remote sequential file access based on measurements of the V file server. Given this model we have investigated the performance of remote file access under increasing load on the file server. We have also explored how high-load performance is affected by various modifications to the file service configuration. We have been able to draw the following conclusions from this modeling study:

1. When viewed in the context of the normal amount of user mode processing, a shared remote file server is quite practical for moderate numbers of workstations, when one is willing to tolerate some amount of

performance degradation in return for the benefits of a shared file server.

2. Under the conditions of the baseline configuration (a CPU comparable to the 68000 and a disk comparable or faster to a Fujitsu Eagle), we have shown that the file server CPU is the bottleneck service center for remote file access. This is true in spite of the fact that we have made rather pessimistic assumptions about the seek vs. I/O ratio and in spite of the fact that we are using communication primitives that are very efficient in terms of processor utilization on the file server.

3. Significant improvements, both in low and high load performance, can be derived from using large size interactions between the client and the file server. For a number of reasons, including buffering limitations, increased probability of packet loss and the small average file size, the rate of improvement drops significantly when the request size increases above 8 kilobytes.

4. Among the modifications of the baseline configuration that we evaluated, introducing a file server cache and introducing a second file server seem most promising in terms of improving performance under high load. They both drastically reduce file server congestion without introducing significant disadvantages.

The queueing network model does not take into account a number of factors. In particular, it only provides us with a first order approximation of the elapsed time distribution, not taking into account the effects of bursty traffic patterns. Simulation and further experience with the file server is necessary to judge whether the predictions of the model are reasonable approximations of reality.

— 5 —
Protocol and Implementation Experience

5.1. Introduction

Having described the communication primitives of the V-System and having studied in detail their performance for various applications, we now turn our attention to the protocol underlying the distributed operation of the V-System. This discussion is divided in two major parts. In the first part we specify the rules of the protocol to which participating machines must adhere if they want to be part of the operation of the distributed V-System. These include rules about allocation of protocol addresses (process identifiers), process location, packet format and valid packet sequences. Motivations for the particular design choices made are outlined. This protocol has been implemented as part of the V kernel to allow workstations to participate in the distributed V-System. Experience with this implementation effort forms the second part of this chapter.

Because of its original implementation in the V kernel, the protocol described in this chapter is frequently referred to as the V *interkernel* protocol. However, the protocol is relatively independent of any particular implementation: any piece of software or hardware implementing this protocol can provide access to the services of a distributed V-System or part thereof. For instance, a process running on a different operating system can implement the protocol and thereby have its clients and its services become part of the V-System. We have currently one example of such a software package in existence: it is made up of a set of Unix processes and makes Unix services available to workstation users[13].

Before starting the discussion of the protocol, we would like to make a couple of disclaimers about the contents of this chapter. First, the description of the protocol and its implementation are slightly idealized versions of the real protocol as it is currently operational. Second, we omit any discussion of other aspects of the V kernel. We refer the interested reader to [7] and [15]. Finally, while for the purposes of discussion, it is nice to draw a sharp line between a protocol definition and its implementation, such a division often becomes blurred in practice, as demands for performance require environment-specific optimizations to be made. The experience with the V protocol in this respect is no different from any other.

5.2. The V Protocol

The discussion of the V protocol is divided in four parts: allocation of V protocol addresses, mapping from V protocol addresses to underlying protocol addresses, packet format and valid packet sequences. We denote by *kernel* the entity that provides the V protocol communication services on a particular machine. This is typically an instance of the V kernel but it might also be some other software package implementing the same protocol. We use the term *process* to denote any communicating entity in the system. Again, this is most often a V process but not necessarily. The protocol addresses in the V protocol are referred to as *process identifiers*. Next, we discuss the allocation of these process identifiers.

[13] It only implements the server side of the protocol, thereby providing access to servers on Unix. It does not implement the client side. Therefore, it is not possible for other Unix processes to become part of the V-System. This restriction is in no way inherent and could easily be removed.

5.3. Process Naming

For the purposes of communicating with other processes, every process needs to have a process identifier that is unique within the V *domain* to which it belongs. A V domain is a set of machines forming a single instance of the V-System. The current implementation assumes that within each domain some form of highly reliable broadcast communication is available. A typical V domain is a (broadcast) local area network.

For reasons of availability and efficiency, we want to allow for distributed generation of process identifiers, while still guaranteeing uniqueness. First, in Section 5.3.1, we present a distributed algorithm for generating unique identifiers -- not necessarily process identifiers -- in a broadcast domain. We analyze this algorithm in Section 5.3.2. Given this analysis, it becomes clear that the proposed method is too costly for direct use in generating process identifiers, given the high frequency of this operation. We then present and analyze a modification of the original method which is better suited to our needs (Section 5.3.3).

5.3.1. Generating Unique Identifiers in a Broadcast Domain

In the method proposed here, the individual kernels cooperate to generate domain-wide unique[14] identifiers in the following way. Let the size of the unique identifier be n bits. Each kernel maintains a record of the identifiers it has currently allocated. When a particular kernel wants to generate a new unique identifier, it picks (at random) an n-bit string and broadcasts it over the domain. All kernels check their records to see if they have that bitstring already in use as a unique identifier. If one of the kernels has that bitstring currently allocated, it transmits a complaint to the requesting kernel. This kernel now tries again with a different n-bit string and continues to do so until it does not receive a complaint within a certain timeout interval after it has announced its choice of unique identifier.

Clearly, the proposed method is not one hundred percent secure. Either the announcement of a new identifier or a subsequent complaint might get lost by the network and as a result two identical unique identifiers might coexist. One can reduce the probability of this happening by executing multiple runs of the algorithm whereby a given identifier is retransmitted for a number of times. Nevertheless, the failure probability remains non-zero. Also, a complaint might be delayed beyond the timeout interval the requesting kernel is waiting for complaints and therefore go unnoticed. Next, we analyze the probability of such a failure occurring and the cost of the algorithm both in terms of running time and communication overhead.

5.3.2. Probability of Failure and Communication Overhead

5.3.2.1 Probability of Failure

We denote by p_f the probability that two identical unique identifiers are generated by the algorithm. In calculating this probability, we make the following assumptions

1. Packet transmissions are statistically independent events. The probability of a packet not arriving at its destination is identical for all packets and denoted by p_n.

2. The timeout interval is long enough so that no "late" complaints have to be considered.

The probability p_f of an undetected identifier collision then becomes the product of the probability p of a collision of unique identifiers, and the probability of either the announcement or the complaint being lost by the network. The probability p is clearly equal to

$$p = m \div 2^n$$

where m is the number of unique identifiers in existence and n is the size (in bits) of the unique identifier. The probability of either the announcement or the complaint being lost is

[14] Uniqueness is defined here as uniqueness at a particular point in time. The issue of preventing identifiers to recycle shortly after they have become invalid is not addressed by this algorithm.

$$1 - (1 - p_n)^2$$

Thus,

$$p_f = (m \div 2^n) \times (1 - (1 - p_n)^2)$$

An estimate must be found for p_n based on the characteristics of the underlying network and the probability of buffer overflow in the network interfaces[15]. Experience with local area networks indicates that this value is typically very low. If necessary, the probability of an undetected identifier collision can be reduced further by retransmitting the announcement of the new identifier several times. If we retransmit α times, and we assume all transmissions are independent, the resulting probability p_f becomes

$$p_f = [(m \div 2^n) \times (1 - (1 - p_n)^2)]^{\alpha}$$

5.3.2.2 Running Time and Communication Overhead

We compute two measures to assess the cost of the algorithm: the expected running time and the expected number of *packet events* caused by one execution of the algorithm. A packet event is defined as the transmission or the reception of a packet. It is indicative of the amount of time the processors in the system spend in executing the algorithm. We assume all other overhead is either negligible or else proportional to the number of packet events.

Denoting by p(i) the probability of success on the i-th try, by T_t the timeout period to wait for complaints, and by T_r the round trip time for an announcement-complaint sequence, the expected running time for the algorithm becomes

$$T_u = T_t \times p(1) + (T_t + T_r) \times p(2) + (T_t + 2T_r) \times p(3) + \ldots$$

Assuming no packets are lost, the probabilities p(i+1) form a geometric distribution with parameter p

$$p(i+1) = p^i \times (1 - p)$$

with p defined as above

$$p = m \div 2^n$$

If the probability of packet loss is taken into account, p needs to be multiplied by a factor $(1 - p_n)^2$. Indeed, in order to have a choice of identifier rejected, the identifier has to be in use already (with probability p) and both the announcement and the complaint must arrive at their destination (with probability $(1 - p_n)^2$).

Summing the above series, we then get

$$T_u = T_t + T_r \times (p \div (1-p))$$

It is clear that for a sufficiently sparse population of the identifier name space (specifically, $p \ll T_t \div T_r$), the value of T_u is dominated by the choice of the timeout interval T_t.

Similarly, denoting by C_u the number of packet events per unique identifier, and by K the number of machines present, we obtain

$$C_u = (K+1) \times p(1) + (2K+4) \times p(2) + (3K+7) \times p(3) + \ldots$$

and, after substitution for p(i) and summation,

$$C_u = (K+1) + (K+3) \times (p \div (1-p))$$

We can also derive the expected time for a second run of the algorithm, i.e. when the announcement is retransmitted in order to reduce p_f. Assuming the same identifier does not get allocated by another kernel between the first and the second run, we get for the expected time for the second run

---

[15] One could extend the above formula by using different failure probabilities $p_{n,bc}$ and $p_{n,uc}$ for broadcast and unicast packets. The probability of either the announcement or the complaint getting lost then becomes $1 - (1 - p_{n,bc}) \times (1 - p_{n,uc})$

$$T_u = T_t + p_f \times T_r \times (1 + p \div (1-p))$$

This formula is arrived at by combining the following two observations. First, with probability $(1 - p_f)$ we have allocated a unique identifier in the first run. In that case, there are no complaints during the second run and after T_t the process terminates. Second, with probability p_f we have allocated a non-unique identifier in the first run. In this case, the original algorithm repeats itself during the second run, except for the fact that there is always at least one complaint (since we assume the identifier is not unique). The average number of announcements is thus identical to that obtained in the first run of the algorithm plus one. Summing the contributions of both cases with the appropriate probabilities, we get

$$T_u = (1 - p_f) \times T_t + p_f \times (T_t + T_r \times (p \div (1-p) + 1))$$

which reduces to

$$T_u = T_t + p_f \times T_r \times (1 + p \div (1-p))$$

For the α-th retransmission, a similar argument gives

$$T_u = T_t + p_f^{\alpha} \times T_r \times (1 + p \div (1-p))$$

Even more so than for the first run, the value of T_u for subsequent runs is dominated by T_t. This is intuitively clear since the probability of a collision is reduced by p_f for each successive run.

5.3.2.3 Alternative Methods

There are a number of modifications one could explore with respect to this basic algorithm. For instance, one could have the different kernels generate random numbers but not do any form of identifier collision detection. Clearly, the cost both in terms of elapsed time as well as in terms of the number of packet events then reduces to zero. The probability of failure can be computed by observing that the probability of two random numbers being equal, in a sample of α from a population of β is approximately equal to

$$(1 - \alpha \div \beta)^{\alpha}$$

or, assuming $\alpha \ll \beta^{1/2}$ and using the binomial approximation,

$$1 - \alpha^2 \div \beta$$

The probability of failure p_f then becomes, substituting m for α and 2^n for β,

$$p_f = m^2 \div 2^n$$

This value has to be compared with the value derived for p_f above for the announcement-complaint algorithm. Whether it is appropriate for a given application has to be judged by substituting the appropriate values of m and n.

Alternatively, one could take advantage of the fact that in most cases the hardware provides a unique host address per machine. The kernel could then use this hardware address as part of the unique identifiers generated by that kernel. Of course, this can only be accomplished if the hardware address (or some unique substring thereof) fits within the size of the process identifier. For instance, on a 10 Mb Ethernet network with its 48-bit addresses, one would have to use process identifiers at least as large as 48 bits. The method would be applicable (and has been used in that form in an earlier implementation of the kernel) on a 3 Mb Ethernet network with 8-bit host addresses.

5.3.3. A Practical Method for Generating Process Identifiers

Using a unique hardware address as part of the process identifier would evidently be the method of choice (virtually zero overhead, perfectly reliable) if it were not precluded by the desire to keep the size of the process identifier small. The method of simply generating a unique identifier without any collision detection leads to unacceptable uniqueness characteristics in our environment. Consider for instance a system with 64 hosts each with an average of 16 processes active: this would produce an error probability p_f of approximately 2.5×10^{-3}. This probability is unacceptably high, in particular if one realizes that it continues to exist over

time. The announcement-complaint method, as described above, is by itself also impractical for generating process identifiers. Given the high frequency of process creation and its relative time-criticalness, the costs in terms of elapsed time and packet events are clearly too high. For instance, the overhead for creating a process -- other than generating its unique identifier -- is on the order of half a millisecond. On the other hand, the elapsed time for generating a unique identifier by the above method is bound from below by the length of the interval T_t during which the kernel waits for complaints. T_t should be at least an order of magnitude bigger than T_r in order to reduce the probability of late complaints, making a value of on the order of 100 milliseconds appropriate for T_t. Clearly, process creation with this method of unique identifier generation would be very expensive.

We remedy this situation by using a traditional hierarchical technique whereby a domain-wide unique process identifier is generated as a combination of a domain-wide unique identifier per machine and a machine-wide unique identifier per process. The method of Section 5.3.1 is used to generate the domain-wide unique *logical host identifier*. This domain-wide unique host identifier is then concatenated with a locally unique identifier to produce an domain-wide unique process identifier. In general, generation of locally unique identifiers can be done trivially in comparison to generating domain-wide unique identifiers (for instance, by taking successive values of a counter). This method has therefore the advantage that the procedure for producing a domain-wide unique identifier has to be invoked only once per machine. When allocating a new process identifier on a particular machine, it is then sufficient to invoke the procedure for allocating locally unique identifiers.

The analysis of Section 5.3.2 remains valid for the generation of unique logical host identifiers. Given the confines of a n-bit identifier, the division of these n bits in h bits for the logical host identifier and n-h bits for the locally unique identifier is a compromise between speedy and efficient logical host identifier generation on one hand, and the desire to have a large space of locally unique identifiers on the other hand (to prevent untimely recycling). In Figure 5-1 we have set out the probability p of an identifier collision, the expected time T_u necessary for generating a unique identifier and its cost C_u of packet events, in terms of h, the number of bits allocated to the logical host identifier field. The expected number of logical hosts is used as a parameter in these curves. From these curves it can clearly be seen that the cost of logical host identifier generation closely approximates its asymptotic value for values of h such that 2^h is slightly bigger than the expected number of logical hosts.

5.3.4. Conclusions

From the analysis in this section, we draw the following conclusions:

1. Process identifiers have to be unique over a V domain. Domain-wide unique identifier generation procedures are too expensive to be invoked on every process identifier generation, given the frequency and the time-criticalness of the latter operation. Therefore, we use a hierarchical technique whereby a process identifier consists of a domain-wide unique logical host identifier concatenated with a locally unique identifier.

2. Several techniques can be used to generate domain-wide unique logical host identifiers. Using a unique address provided by the hardware would be the method of choice if it were not precluded by considerations regarding the relative size of process identifiers and host addresses. Therefore, an announcement-complaint algorithm can be used.

3. Single run announcement-complaint algorithms have some probability p_f of failing, the value of p_f being dependent on the population of the identifier space and the error characteristics of the network. The cost of a single-run algorithm is in practice dominated by the timeout interval T_t. The failure probability can be reduced (with a factor of p_f per run) by using a multiple-run algorithm, at the expense of another T_t in runtime for each additional run.

4. For efficient logical host identifier generation, it is sufficient to allocate to the logical host identifier field h bits, such that 2^h is slightly bigger than the number of logical hosts in existence.

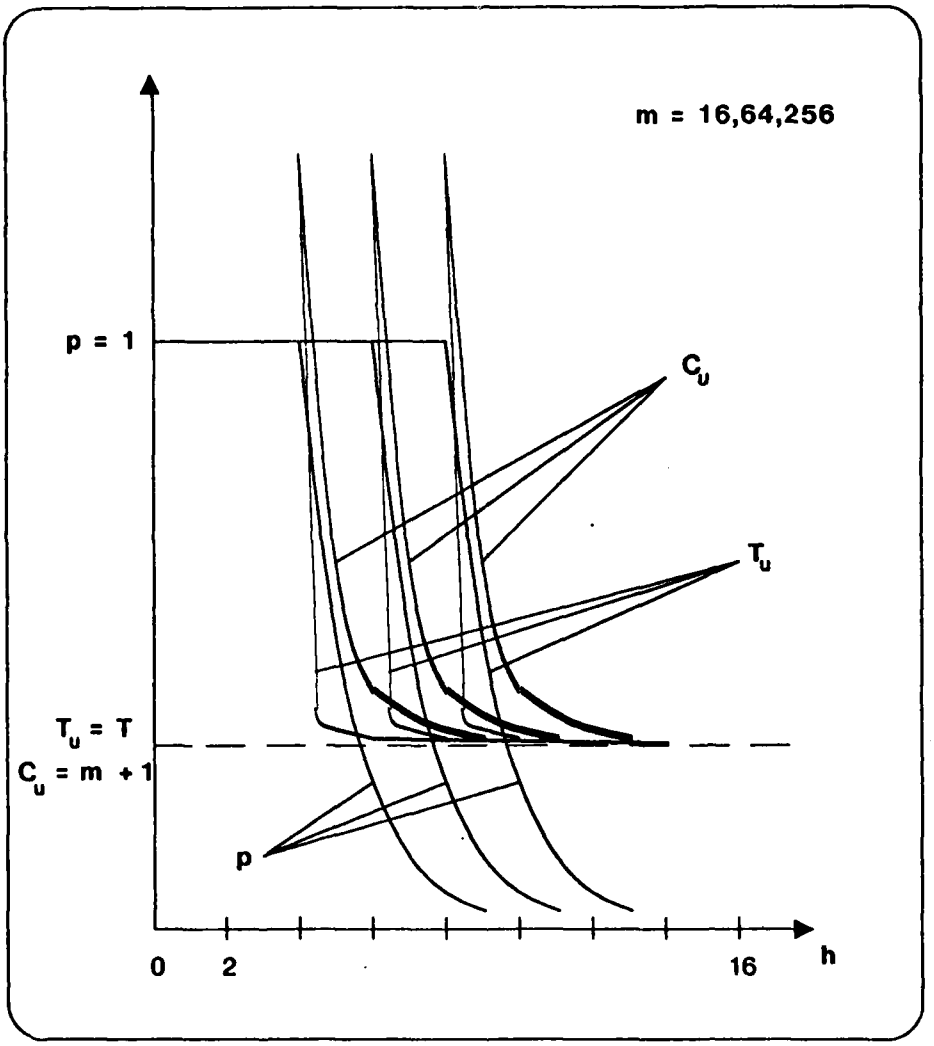


Figure 5-1: Uniqueness Characteristics

In practice, we use 32-bit process identifiers. This is the natural word size of the machines currently used in the V environment. The high-order 16 bits of the process identifier serve as the logical host identifier subfield. During initialization, the kernel picks a random 16-bit number by looking at its machine clock[16]. It then executes a single run of the announcement-complaint algorithm, which results in satisfactory uniqueness guarantees for our environment. The timeout period T_t is set at 0.5 seconds.

5.4. Process Location

In order to have a message sent from a process on one machine to a process on another machine, the kernel on the sender's machine must format a packet and instruct the underlying protocol layer to transmit that

[16] A subtle point: the machine clock and not a software clock. This results in a much more random result after rebooting, as desired.

packet to the appropriate machine. To do so, it must translate the process identifier of the target process into an address that is meaningful to the underlying protocol layer. It follows from our assumptions about a V domain that the underlying protocol layer provides a broadcast address, that is, a special address by which all kernels can be reached. We now discuss *process location*, this is the method by which a kernel translates a process identifier into an underlying address. In the current implementation of the V-System, the V protocol is implemented on top of a data link protocol and therefore the underlying protocol addresses are referred to as host addresses, and the communicating entities as hosts. However, this is not essential to this discussion.

5.4.1. Process Location by Broadcasting

The simplest process location strategy would be to have none at all. Given the fact that we assume that some form of broadcast is available in each V domain, every packet could be broadcast. All kernels would see all messages and filter out those destined for their processes. For our environment, we believe this approach to be impractical, because it requires all kernels to process all messages (regardless of their destination). The cost C_m in terms of packet events of sending a message would be equal to the number of kernels present plus one (The "plus one" derives from the fact that the sending kernel listens to its own transmissions, resulting in 2 packet events on the sending kernel.)

$$C_m = K + 1$$

This would result in an unacceptably high load on the processors for protocol handling. This approach might become practical if part or all of the (network) interprocess communication machinery could be offloaded to a smart network interface processor, and if the broadcast traffic does not interfere with the operation of machines in the broadcast domain that are not participating in the V-System[17]. One would presumably still have to worry about a number of issues, including greatly increased queue lengths at the network interface, and, as a result, an increased probability of packet loss.

In order to reduce the overhead associated with network interprocess communication, the protocol provides a more sophisticated location procedure which attempts to deliver most messages by point-to-point communication, thereby reducing C_m to 2 per message.

5.4.2. Efficient Process Location

Conceptually, process location can be accomplished by providing a global table ***map***, mapping process identifiers into host addresses. For the time being, we assume *strong* semantics for the entries in this table, that is, we assume that a process with process identifier ***pid*** can only exist at the host with host address ***map[pid]***[18]. Clearly, for reasons of availability and efficiency, the idea of such a global table is impractical from an implementation viewpoint. In order for kernels to be able to make process location decisions based solely on local information, we wish to replicate the *map* table over all kernels. It is not necessary for correct operation that each kernel maintains a full copy of the table. Indeed, if it does not possess an entry for a particular process identifier, the kernel can resort to broadcasting messages directed to that process. However, it is essential that when an entry is present for a process identifier, the corresponding host address is correct, i.e. that the process actually exists at that host address (or does not exist at all). Otherwise, the kernel would erroneously conclude that that process identifier is invalid.

Given the fact that a particular kernel maintains only an *incomplete view* of the mapping table, there is some probability q that the host address of a process can be derived from that view. On average, then, the packet event cost for sending a message is

$$C_m = 2 \times q + (K + 1) \times (1 - q)$$

---

[17] This could potentially be accomplished by *multicasting*: this is addressing packets to a subset of all machines in the broadcast domain.

[18] Alternatively, the table entries could be *hints*.

The probability q is a function of the size of the view relative to the overall size of the name space for process identifiers as well as of the strategy used to decide what mappings to store in the view. An LRU strategy seems appropriate since one expects some locality of reference with respect to process identifiers.

The space requirement in the kernel for maintaining a view of size S, with process identifiers of size n and host addresses of size a, is then

$$space = S \times (n + a)$$

In order to achieve a reasonably low average value of C_m, it might be necessary to maintain a relatively large view of the map, and the corresponding space demands in the kernel might be significant. This situation can be improved by observing that the number of different host addresses (machines) in a V domain is typically small (with respect to the possible number of process identifiers, 2^n). Therefore, it is attractive to divide up the process identifier name space into a set of equivalence classes, with each equivalence class mapping to a single host address. Then, only a single entry per equivalence class is necessary in the mapping table. The kernels deduce the equivalence class from the process identifier by encoding the equivalence class in a substring of the process identifier. All process identifiers with the same encoding substring then reside on the same host (assuming they exist at all) and they are all represented by a single entry in the map, mapping that substring into a host address. In this way, the size of the view a kernel has to maintain in order to achieve a certain value of C_m can be reduced significantly. The formula becomes

$$space = S \times (h + a)$$

where h is the size of the host encoding substring. The value of S can be chosen much smaller in this case because

1. The size of the name space is much smaller (2^h vs. 2^n)
2. The locality of reference with respect to hosts is far more pronounced than the locality of reference with respect to process identifiers, resulting in a more effective utilization of the table.

In practice, the 16-bit logical host identifier subfield of the process identifier serves as a host encoding substring. The different kernels maintain a table mapping some subset of these logical host identifiers into host addresses. New entries are entered in the table as new mappings are inferred from incoming packets[19]. Old entries are deleted on an LRU basis. If a message has to be delivered to a process for which the table does not contain a mapping, the message is broadcast.

5.4.3. Process Migration

We define *transparent process migration* as the operation whereby a process moves from one physical host within the V domain to another, in such a fashion that processes that communicate with it do not have to be aware of its change in location. A full discussion of process migration within the framework of the V-System is beyond the scope of this thesis. The following discussion contains some observations, relating solely to the subject of process location[20].

If all V messages were transmitted by broadcast, as suggested in Section 5.4.1, then process migration would not cause extra problems with respect to process location. All kernels receive all messages and a message directed to a particular process is delivered without problem even if that process has just moved from one host to another. However, in order to reduce the overhead associated with interprocess communication, we wish to deliver most messages by point-to-point communication. Kernels must now be able to associate a particular host address with a process identifier. If processes are allowed to migrate, the database containing these associations must somehow be able to reflect the changes. Two solutions seem plausible. The first solution consists of updating the database. This is severely complicated by the fact that the database is

[19] Incoming packets contain both the source process identifier and the source host address.

[20] A full discussion of process migration in the V-System will be the subject of Marvin Theimer's forthcoming Ph.D. thesis.

distributed, and more specifically (partially) replicated over an *unknown* number of kernels. Broadcasting the update some number of times would provide good (statistical) certainty that all tables have been updated. The alternative solution requires changing the semantics of the entries in the database and treating them as hints. In that case, no attempt is made to update the database when a process moves. However, rather than assuming that a process does not exist if it is not found at the host address corresponding to its table entry, the kernel in that case falls back on broadcasting the message (or falls back on transmitting a message to another location if it receives some form of *redirect* message).

Both methods have their costs and benefits. Broadcasting the update α times results in a cost of

$$C_{move} = \alpha \times (K + 1)$$

packet events per process that migrates. Treating the mappings in the table as hints causes the first message from a particular kernel to that process after its move to cost

$$C_m = \beta + (K + 1)$$

in packet events (assuming β retransmissions and no redirect messages). More importantly, it causes that first message exchange to last for the normal message exchange time plus the timeout interval for message exchanges.

Also, establishing invalidity of a process identifier becomes more expensive when hints are used. When the mapping is known to be correct, it suffices to deduce the host address using the table and to check the validity of the identifier at that host address. This results in a fast (in)validity check. If the table contains only a hint, invalidity is harder to establish. Indeed, one can now no longer conclude from the non-existence of the process at the hint host address that the process does not exist at all. In the best case, the kernel on that machine is able to provide some form of *redirect* message, or in the worst case we have to fall back on the broadcast mechanism.

The fact that processes are allowed to migrate away from the host on which they were created has a few other minor repercussions. First, if this were not to be the case, all processes on a particular machine would have the same logical host identifier subfield in their process identifier. This is now no longer true, resulting in a slightly more complicated check for locality. This is important since this check is performed on every interprocess communication operation. Second, it is no longer true that all process identifiers of processes on a particular host are allocated by a single kernel. If this were the case, then some subfield of the local unique identifier could be kept unique at any particular time, resulting in a trivial hashing function into the process descriptor table. Finally, if some form of host encoding is done, and the encoding has strong semantics, then all processes with the same host encoding must, by definition, be present on the same machine and consequently move all at the same time.

5.4.4. Conclusions

Given adequate hardware support in the network interfaces, process location by broadcasting would be the method of choice in a broadcast domain. Given the limitations of current hardware, we need to resort to point-to-point communication to reduce the overhead of network interprocess communication. We have shown a simple procedure whereby each kernel maintains an incomplete view of the mapping of process identifiers to host addresses and resorts to broadcasting only when the mapping for a particular process identifier is not locally available. In order to improve the hit ratio as well as to decrease the size of the view each kernel must maintain, we use host encoding in the process identifiers. We have documented the repercussions of point-to-point communication with respect to process migration. Two solutions were suggested: either updating the views of the different kernels or treating the entries in the views as hints.

5.5. Packet Layout

The interkernel packet consists of two main portions: a fixed size header portion containing protocol related information and the message itself, and a variable size data portion, meant to carry the data in a ***MoveTo*** or

MoveFrom, or the data appended to a ***Send*** or a ***Reply***. The size of the header is 68 bytes[21]. The size of the data portion may range from zero bytes to the maximum allowed by the given network.

The header and message portion are fixed length. Not all the fields are used for every packet. However, the fixed size allows for efficient handling of network packets through DMA network interfaces (See Section 5.7.3). The ease with which a DMA transfer can thus be set up offsets the disadvantage of having to put some number of extra bytes on the network for every packet. For instance, for a page write of 1024 bytes, the fixed header size requires 28 extra bytes to be sent, resulting in an extra overhead of 60 microseconds (on the 10Mb network using a 3-Com interface) on a total of 7.5 milliseconds. For a ***MoveTo***, there are 36 extra bytes in every packet. For a 64 kilobyte ***MoveTo***, this results in an extra overhead of approximately 5.2 milliseconds on a total of 200 milliseconds.

The detailed packet format is shown in Figure 5-2. Most fields have the same meaning for all packet types; they are listed below. A number of fields have specific meanings depending on the packet type; they are discussed as the individual message sequences are examined (See Section 5.6).

packetType Indicates the type of operation being performed. Possible values are ***logicalHostRequest***, ***logicalHostComplaint***, ***remoteSend***, ***remoteReply***, ***remoteForward***, ***nAck***, ***replyPending***, ***remoteGetPid***, ***remoteGetPidReply***, ***remoteMoveToRequest***, ***remoteMoveToReply***, ***remoteMoveFromRequest*** and ***remoteMoveFromReply***.

sequenceNumber Sequence number of the message exchange. The sequence numbers need only be unique relative to each sending process, although in practice it is unique relative to each network node.

sourcePid Process identifier of the process that executed the corresponding operation.

destinationPid Process identifier of the destination process; zero in the case of ***remoteGetPid***.

forwarderPid Process identifier of the process that forwarded this packet.

userNumber User number of the source process. This is currently unused. It is intended to be used for security and authentication purposes.

length Length in bytes of the data segment appended to this packet.

totalLength Total length of the data segment of which this packet is part.

localaddress Originating address of the segment data.

remoteaddress Destination address of the segment data.

The V protocol packet is to be encapsulated in the packet format of the underlying protocol. It is assumed that this underlying protocol provides some form of *type* field by which it is possible to distinguish V packets. Current implementations of the V protocol are built on the 3 Mb and the 10 Mb Ethernet data link protocol. Consequently, V packets are encapsulated in Ethernet data link packets for transmission.

[21] Not counting any network-level or data link-level encapsulation

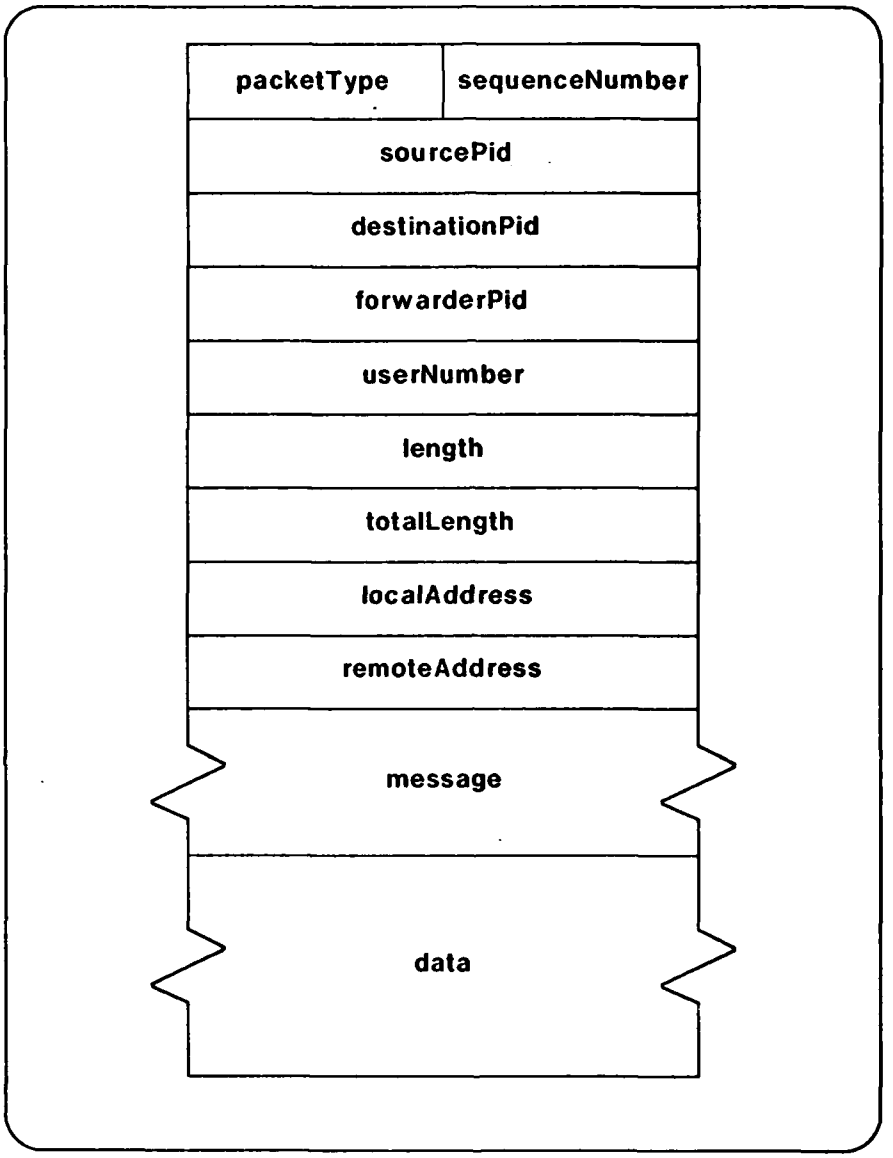


Figure 5-2: Interkernel Protocol Packet Format

5.6. Packet-Level Communication

5.6.1. Remote Message Communication

This section discusses the packet exchanges necessary to perform a remote ***Send-Receive-Reply*** sequence. We first describe the packet sequence under normal conditions. Exceptional conditions are discussed in the next paragraph.

5.6.1.1 Normal Sequence of Packets

The kernel on the sending machine transmits a ***remoteSend*** packet on the network, either directly to the machine on which the destination process resides or else by means of a broadcast packet. The relevant fields in the packet (See Figure 5-2) are ***sequenceNumber, sourcePid, destinationPid, forwarderPid***[22] and ***message***. The ***length*** field is zero. Assuming the receiving kernel has a message buffer available, it accepts the incoming message and queues it for the destination process. This does not cause any acknowledgement packet to be sent at this time. When the receiving process replies to the message, a ***remoteReply*** packet is sent with ***sequenceNumber*** identical to that of the original ***remoteSend*** packet, and furthermore with the values of ***sourcePid, destinationPid*** and ***message*** set to the appropriate values. The ***length*** field is again zero. Again, there is no explicit acknowledgement to the ***remoteReply*** packet.

5.6.1.2 Exceptional Conditions

In order to deal with lost packets or deceased hosts and processes, the protocol provides a retransmission strategy. We first discuss the sending kernel's strategy and then the receiver's.

When the original ***remoteSend*** packet is fired off, the sending kernel starts two timers associated with this message, a *retransmission* timer with interval T_r and a *timeout* timer with interval T_t. If the retransmission timer expires, a ***remoteSend*** packet identical to the original is retransmitted and the retransmission timer is restarted. If the timeout timer expires, the destination process is deemed dead or unreachable, the message exchange is terminated, and an error code is returned to the sending process.

When a ***remoteSend*** packet comes in, the kernel compares the (***sourcePid, sequenceNumber***) pair in the incoming packet against the list of such pairs of message exchanges currently in progress. If the packet is a retransmission, the kernel discards the packet and sends back a ***replyPending*** packet. It also sends back a ***replyPending*** packet if it is forced to discard a new message because no buffers are available. Such a ***replyPending*** packet causes the retransmission timer and the timeout timer to be restarted. This prevents a ***Send*** from timing out due to temporary unavailability of buffers or due to lengthy processing at the receiver.

Additionally, messages can be marked as being idempotent or non-idempotent requests. For an idempotent request, all notion of the message exchange is discarded by the receiver's kernel as soon as the ***Reply*** is executed. If the ***remoteReply*** packet gets lost or if a retransmission of a ***remoteSend*** occurs in parallel with the transmission of the ***remoteReply*** packet, this can cause the message to be delivered twice (or more). Since the operation is marked as idempotent, this should not cause any harm. If the message is marked non-idempotent, the kernel keeps around the ***Reply*** message for an amount of time at least as big as the retransmission time T_r. If a retransmission of the ***remoteSend*** packet comes in, the message is not delivered to the receiver but the ***Reply*** is returned instead. The next message from the same process causes the ***Reply*** to be deleted as well, since this indicates that the message exchange terminated on the sender's end (either by receiving the ***Reply*** or by timing out). This next message then effectively functions as an acknowledgement of the ***remoteReply*** packet.

When a segment needs to be appended to the ***Send***, the corresponding ***remoteSend*** packet's ***length*** field indicates the size of the data segment appended to the message. The same applies for a ***remoteReply*** with a segment appended. Additionally, for the latter the ***remoteAddress*** field indicates the origin of the segment in

[22] Identical to ***sourcePid***

the sender's address space where the data is to be put.

Finally, when a process executes a ***Forward*** on a message coming from a remote process, the kernel sends back a ***remoteForward*** packet to the kernel of the original sender. A ***remoteForward*** packet is identical to a ***remoteReply*** packet except for the packet type. The sender's kernel then transmits a new ***remoteSend*** packet, identical to the original one, except for the ***destinationPid*** and ***forwarderPid*** fields which are modified appropriately.

5.6.1.3 Examples

Thus, in a typical scenario, a ***remoteSend*** and a ***remoteReply*** packet are exchanged for a ***Send-Receive-Reply*** exchange. When the receiver does not ***Reply*** within the time interval T_r, the ***remoteSend*** is retransmitted and a ***replyPending*** is sent back in return at time intervals T_r until the ***Reply*** is sent. If the receiver does not exist, a ***remoteSend*** packet is transmitted a number of times at intervals T_r until the timeout interval T_u expires. As a performance optimization for this case, if a kernel is present on the target machine, this kernel sends back a ***nAck*** packet to quickly terminate the message exchange. Finally, if a packet gets lost, either the ***remoteSend*** or the ***remoteReply***, the ***remoteSend*** is retransmitted, causing the message exchange to be reexecuted.

5.6.1.4 Discussion

The protocol described above is an instance of a *reliable message transport* protocol. The underlying layer is an unreliable datagram protocol. The gap between these two protocols is typically bridged by a PAR (positive acknowledgement and retransmission) strategy. The sender transmits a packet until it is acknowledged by the receiver and so does the receiver for the reply packet, leading to four packets for the overall exchange. However, due to the request-response nature of the V interkernel protocol, i.e. due to the fact that every request has a response associated with it, it is possible to reduce the number of packets in the above sequence to two. This is accomplished by having the reply message function as the acknowledgement to the original request.

It is also necessary for the sender to periodically check whether the receiver is still alive, causing two packets to be exchanged at regular intervals T_r. The advantages of the request-response nature of the protocol are thus most significant when the message exchange is short. For instance, if the message exchange completes within T_r, then only two packets are necessary, as opposed to four when each individual packet were acknowledged. The main drawback of this strategy stems from the fact that the ***remoteReply*** is not explicitly acknowledged. When the ***remoteReply*** is lost, the ***remoteSend*** gets retransmitted. In the case of an idempotent request, this causes the whole operation to be reexecuted. For non-idempotent requests, it requires that state about the request (the reply message and its sequence number) be kept around for a longer time than would typically be necessary if the ***remoteReply*** were explicitly acknowledged.

The choice of T_r is governed by a compromise between speedy detection and correction of communication failures on one hand and the desire not to expend large amounts of processor time (and to a lesser extent network bandwidth) in unnecessary retransmissions. Assuming independent failures, the number of retransmissions forms a geometric distribution with parameter q, whereby q is the probability of a message exchange completing without lost packets

$$q = 1 - (1 - p_n)^2$$

The expected value and the standard deviation for the message exchange time is then

$$\mu = T_0 + (T_0 + T_r) \times (q \div (1-q))$$
$$\sigma = (T_0 + T_r) \times (q^{1/2} \div (1-q))$$

where T_0 is the message exchange time when there is no failure. For typical values of p_n, which are on the order of 10^{-6} for a local network, the retransmission interval T_r has negligible effect on the expected time but is the dominant term in the standard deviation.

Arguing against a small retransmission interval is the desire to reduce overhead associated with network interprocess communication. For instance, consider a message containing a request whose processing takes T

seconds. Then, the number of retransmissions of this message is $T \div T_r$. Retransmissions of the message cause the processing of the request to be interrupted for receiving the retransmission and generating the ***replyPending*** packet. In the current implementation, this requires slightly over a millisecond in processing time. Thus, due to retransmission handling, the actual elapsed time for the request becomes

$$T + 10^{-3} \times (T \div T_r) = T \times (1 + 10^{-3} \div T_r)$$

For a retransmission interval of 100 milliseconds for instance, the elapsed would be inflated by one percent.

Due to the low error rates and due to the very efficient coding of the primitives, the retransmission interval has a relatively limited effect on the message passing efficiency. It has only a second order effect on the elapsed time due to the low error rates, and, due to the very efficient coding of the primitives, it has only limited effect on the processor utilization. We are currently using a retransmission interval of 2.5 seconds. The reason for the long retransmission interval is primarily the desire to communicate with the V simulator running on a loaded, timeshared VAX/UNIX system.

5.6.1.5 Idempotency and At-Least-Once Semantics

An important consideration in the design of a reliable transport protocol is the choice of delivery characteristics guaranteed by the protocol in the face of communication and machine failures.

Let us first restrict the discussion to communication failures. A relatively simple strategy is to provide *at-least-once* semantics. The protocol hereby guarantees that if a reply is returned to the sender, the message has been delivered at least once. This is simply accomplished by retransmitting the message some number of times until a reply is received. If no reply is received in time, an indication of no delivery is returned[23].

A more sophisticated strategy is *exactly-once*. In this case, a reply indicates that the message was delivered exactly once, with again an indication of failure if no reply is received within some interval. On top of periodically retransmitting, as for at-least-once, exactly-once requires maintaining a history of the identification numbers (typically a source process identifier and a sequence number) of received messages and suppressing duplicates by comparing the identification number of incoming messages against the history. Additionally, it requires that replies and identification numbers be kept around for some amount of time after the ***Reply*** has been executed until it is certain that no retransmissions will come in. It is necessary to keep around this information in order to be able to return the reply when a retransmission comes in.

The V-System supports both at-least-once and exactly-once semantics: the sender indicates in the message the desired semantics. Cedar RPC supports exactly-once semantics [9]. The cost of providing exactly-once semantics primarily results from the need for the kernel to make a copy of the ***Reply*** in order to keep it around for potential retransmission. The amount of data associated with a ***Reply*** can be quite substantial, for instance in the case of a page read operation. Not supporting exactly-once communication semantics requires the applications to implement *idempotent* operations, i.e. operations whose side-effects and return values are independent of whether they were invoked once or more. More experience is needed to make a well-founded judgement about this trade-off. On one hand, most applications desire exactly-once semantics. It is therefore appealing to implement it once in the communication layer and then provide it as a service to applications. On the other hand, for certain applications, implementing idempotent operations is far less expensive than using an exactly-once communication primitive with its need for an extra copy. For instance, a file system page-level read can be made idempotent without any extra effort: the page number provides a unique identifier that can be used for detecting duplicates and the data associated with the read is always available later since it is stored in the file system.

Neither the V-System nor Cedar RPC provide any guarantee of delivery across machine crashes. It is assumed that clients explicitly rebind to servers and execute some appropriate protocol to recover from

---

[23] The indication of non-delivery is only correct with high probability. The last transmission of the message could have succeeded and the reply to that message could have failed or could have been delayed beyond the timeout interval. The retransmission and timeout strategy should be such that this possibility is exceedingly unlikely.

machine failures. In contrast, Liskov's RPC guarantees what is called *at-most-once* semantics in [48], even across crashes. When a reply is returned, it is guaranteed that the message has been delivered exactly once. Otherwise, an indication of failure is returned and it is guaranteed that the message has had no effect. This makes the remote message invocation effectively a multi-machine transaction, with associated commit and abort protocols. In particular, to survive machine crashes, message transactions have to be recorded on stable storage. The cost of doing so is quite clear, at least with today's technology. Given a cost of 2 to 3 milliseconds per message, and a cost of on the order of 20 milliseconds to access stable storage (a pair of disks, for instance), the overall cost of a message transaction increases by an order of magnitude. Development of fast stable storage, such as battery backed-up core memory, could drastically alter the economics of this situation.

5.6.1.6 Conclusions

For remote message exchanges, we use a protocol that takes advantage of the low-error, low-latency characteristics of local networks and of the expectation that the duration of most message exchanges is short (relative to the retransmission interval). Under those assumptions of low error rate, low latency and short message exchanges, the actual value of retransmission interval used has only secondary effects on the perceived performance of message passing, both in elapsed time and in processor utilization. Also, we have shown the implications of different reliability characteristics of message delivery. In particular, we have identified the significant cost of exactly-once semantics and explored the alternative of providing idempotent operations.

5.6.2. Remote Data Transfer

5.6.2.1 Error-Free Packet Exchanges

When executing a ***MoveTo***, the kernel breaks up the total segment in a number of maximally-sized packets. It transmits these in a sequence of ***remoteMoveToRequest*** packets without ever waiting for an acknowledgement. Besides ***sourcePid***, ***destinationPid*** and ***sequenceNumber***, the relevant fields in the packet are ***length***, ***totalLength***, ***localAddress***, and ***remoteAddress***. ***length*** contains the length of the data segment appended to this particular packet, ***totalLength*** indicates the total length of the ***MoveTo*** operation. ***localAddress*** indicates the start address of the remote data segment, while ***remoteAddress*** indicates the destination address of the data in this particular packet. When the whole sequence of packets has arrived at the destination machine, the kernel on the destination machine sends back a ***remoteMoveToReply***.

MoveFrom works in a similar fashion except that the kernel executing the ***MoveFrom*** sends a single ***remoteMoveFromRequest*** packet to the destination kernel with ***totalLength*** indicating the total length of the ***MoveFrom***. ***localAddress*** contains the address of where the process expects the data and ***remoteAddress*** indicates where it is to come from. In response, the destination kernel breaks up the segment in maximally-sized ***remoteMoveFromReply*** packets and transmits them to the source kernel without ever waiting for an acknowledgement. The relevant fields in the ***remoteMoveToReply*** packets are ***length***, indicating the length of the data segment appended to this packet and ***remoteAddress*** indicating where to put the data in this packet in the address space of the process executing the ***MoveFrom***.

5.6.2.2 Dealing with Errors

In order to deal with lost packets, a number of different strategies can be considered. In the current implementation, the destination kernel of the ***MoveTo*** only sends an acknowledgement when it has received all packets. If the sender does not receive an acknowledgement within a retransmission interval T_r, it retransmits the whole sequence. Given the low error rates of local area networks, full retransmission on error introduces only a slight performance degradation. However, full retransmission can cause a ***MoveTo*** to fail repeatedly if back-to-back packets are consistently being dropped by the receiver. Also, full retransmission to deceased processes causes substantial amounts of processor time and network bandwidth to be wasted. In order to avoid these problems, we are investigating a more elaborate, selective retransmission strategy. The

receiver waits until it sees the last packet in the sequence. At this point, it returns a bit vector[24] indicating the packets which it successfully received. The sender then retransmits the missing packets. If no acknowledgement is received within T_r, the last packet in the sequence is retransmitted, thereby eliciting an acknowledgement from the receiving machine.

5.6.2.3 Discussion

In this section we analyze the expected time and the standard deviation for executing a ***MoveTo*** operation under different transmission strategies. ***MoveTo*** operations are different from other forms of large-scale data transfer that have been analyzed in that, by definition, the recipient of the data has sufficient buffers allocated to receive the data. Additionally, receipt of packets belonging to a ***MoveTo*** is handled with the highest priority (at the network interrupt level) and is as a result not slowed down by process scheduling delays. These two factors, combined with the fact that the speed of the sender and the speed of the receiver are more or less matched, allow the ***MoveTo*** operations to be implemented in a way that takes maximal advantage of the high speed of local area networks without worrying about time-consuming flow control measures. As such, the protocols presented here represent techniques for achieving near-to-optimal performance on a local network. In this analysis, we again assume that packet transmissions are statistically independent events which can fail with probability p_n. Since some of the derivations in this section are quite lengthy, we state the results of the analysis up front in the next paragraph. The derivation of these results follows.

We are considering a number of different transmission strategies, including

1. Stop-and-wait: acknowledge every packet.
2. Streaming (a single acknowledgement for all packets) with full retransmission on error, with or without negative acknowledgement.
3. Streaming with selective retransmission.

We show that for error rates typical of local area networks, the *expected time* for a ***MoveTo*** under a given transmission strategy is almost exclusively dependent on the elapsed time for that strategy when no errors occur, and relatively independent of the retransmission strategy used. Therefore, any strategy that is suboptimal in the no-failure case also has suboptimal performance under realistic local network operating conditions. This is not unexpected given the low error rate; our contribution is that we quantify this intuitive insight. Second, we show that, again for typical local network error rates, the standard deviation is dependent on the retransmission strategy, in particular on the amount of data retransmitted and the retransmission interval. The effects of various modifications to the retransmission strategy are quantified.

5.6.2.4 Expected Time

In this section we analyze two strategies that have significantly different no-failure characteristics, namely stop-and-wait and streaming with full retransmission on error and no negative acknowledgement. We show that for typical local network error rates, both strategies have the same expected time as in the no-failure case. Therefore, evidently, the streaming strategy shows much better results than the stop-and-wait strategy.

The analysis of the stop-and-wait protocol is very similar to the analysis for message exchanges. Denoting by $T(D)$ ($T(1)$) the time necessary for a D-packet (1-packet) transfer, we obviously have

$$T(D) = D \times T(1)$$

The probability of a 1-packet exchange failing is

$$q = 1 - (1 - p_n)^2$$

and the probabilities $q(i+1)$ of the exchange succeeding on the i-th retransmission form a geometric distribution with parameter q

[24] It is suggested the message field in the packet is used for this purpose.

$$q(i+1) = q^i \times (1 - q)$$

The expected time for a 1-packet transfer to complete is then

$$T_0(1) + (T_0(1) + T_r) \times (q \div (1-q))$$

where $T_0(1)$ is the time necessary for a 1-packet exchange without any errors. For D packets we get for the expected time

$$\mu = D \times [T_0(1) + (T_0(1) + T_r) \times (q \div (1-q))]$$

Let us next consider the case of full retransmission on error (without a negative acknowledgement). A D-packet transfer succeeds in this case if all D packets reach the destination machine and the acknowledgement packet reaches the source machine. Assuming independent transmissions, the probability of the D-packet transfer failing is then

$$q = 1 - (1 - p_n)^{D+1}$$

Given this probability q, the probabilities q(i+1) that a ***MoveTo*** attempt succeeds on the i-th retransmission form a geometric distribution with parameter q

$$q(i+1) = q^i \times (1 - q)$$

The expected time T(D) for a D-packet transfer becomes

$$\mu = T_0(D) + (T_0(D) + T_r) \times (q \div (1-q))$$

Figure 5-3 compares the two strategies for different values of T_r (The other parameters in the figure are D = 64, $T_0(1)$ = 5.8 msec. and $T_0(D)$ = 198 msec., the latter two from experimental measurements.) Additionally, Figure 5-4 shows the effect of breaking up very large data transfers in different size of ***MoveTos:*** shown in the figure is a 512 kilobyte transfer, broken up in 8, 64 and 512 kilobyte operations.

Clearly, for typical values of p_n (on the order of 10^{-6}), the expected time for both strategies is nearly identical to the no-failure expected time and relatively independent of the actual value of T_r. The retransmission interval only affects the location of the knee in the curve which falls outside the domain of typical values of p_n. In comparing the results for the stop-and-wait protocol and the streaming protocol, the key observation to make is that $T_0(D)$ -- the no-failure value for the streaming protocol -- is significantly smaller than D × $T_0(1)$ -- the comparable value for the stop-and-wait protocol (See Section 3.4.3). Consequently, for low error rates, where this term dominates, the streaming strategy performs significantly better. There remains the question of what the optimal streaming size should be: Figure 5-4 shows that the improvement in no-failure behavior becomes small as the streaming size is increased above 8 kilobytes. Smaller sizes are evidently also more robust against higher failure rates[25].

These results also allows us to make a stronger conclusion: since the expected time for streaming with the crudest retransmission strategy -- full retransmission on error and no negative acknowledgement -- results in a nearly optimal expected time (for the appropriate range of p_n values), no significant improvements in expected time can be achieved by more sophisticated retransmission strategies. In the next section, we show that such strategies can significantly improve the standard deviation.

5.6.2.5 Standard Deviation

Given the results of analyzing the expected times, we consider in the rest of this discussion only strategies that have optimal no-failure characteristics. We also assume that we operate under such error conditions that the expected time of the transfer is nearly identical to the no-failure transfer time (i.e. we operate in the leftmost region of the curves in Figure 5-3). We now analyze the standard deviation of different strategies.

Consider a given transmission strategy and denote by $T_0(D)$ the elapsed time in the no-failure case.

[25] In Chapter 4 we have already indicated a number of reasons why using larger interaction sizes between a file server and its clients does not result in significant gain once the interaction size is increased beyond 8 kilobytes.

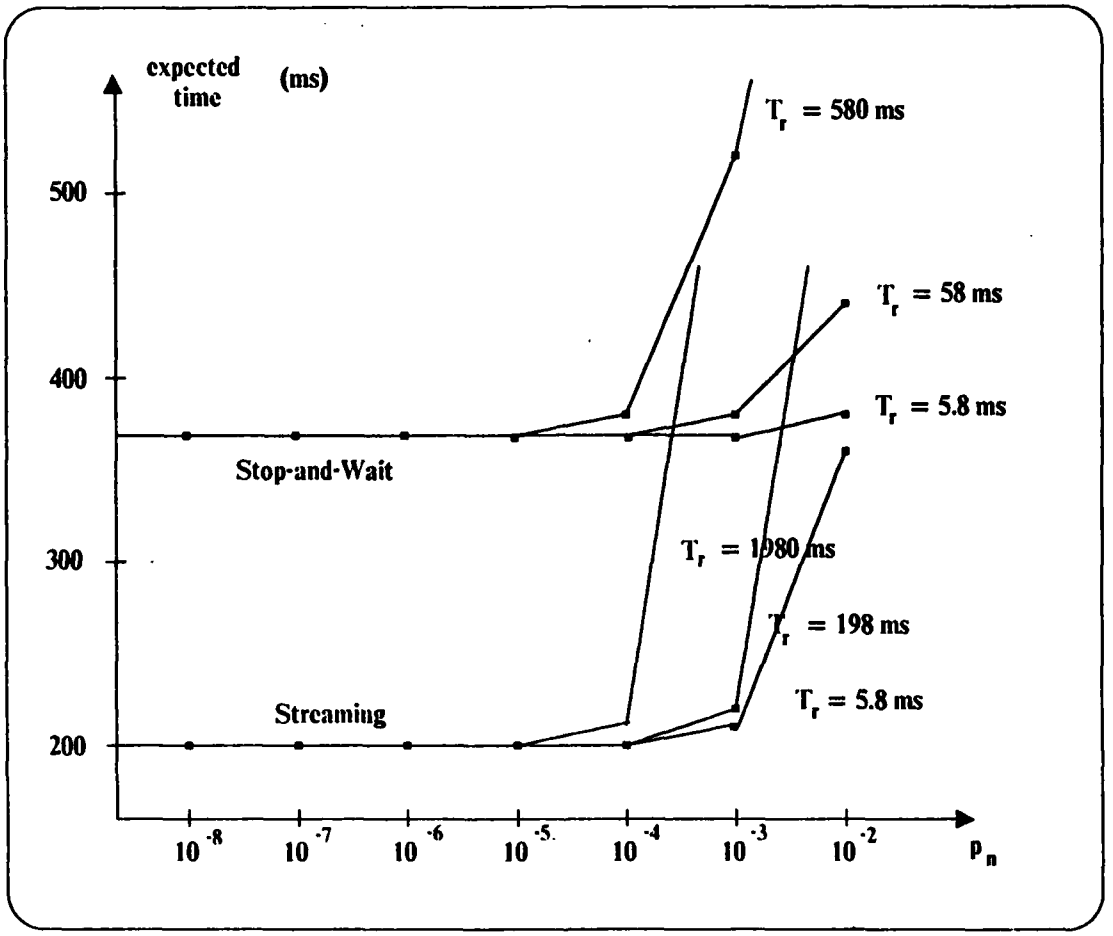


Figure 5-3: Expected Time for 64 kilobyte Transfers

Furthermore, let $T_0^{(k+1)}(D)$ be the elapsed time for the k-th retransmission, let $T_r^{(k+1)}(D)$ be the interval interval between the k-th and the (k + 1)-th retransmission, and finally let q(i + 1) be the probability of success on the i-th retransmission. Then, if the transfer succeeds on the i-th retransmission, the total elapsed time for this transfer is

$$\Sigma_{k=0}^{k=i} T_0^{(k+1)}(D) + \Sigma_{k=0}^{k=i-1} T_r^{(k+1)}(D)$$

Assuming we are operating under low error conditions and that thus the expected time is constant and approximately equal to $T_0(D)$, we get for the variance

$$\sigma^2 = \Sigma_{i=0}^{i=\infty} [(\Sigma_{k=0}^{k=i} T_0^{(k+1)}(D) + \Sigma_{k=0}^{k=i-1} T_r^{(k+1)}(D))^2 \times q(i+1)] - T_0^2(D)$$

This formula indicates three potentially fruitful avenues for reducing the variance:

1. Reduce the retransmission intervals $T_r^{(k+1)}(D)$: this can be accomplished either by choosing a small timeout value or by providing a negative acknowledgement when the transfer fails.

2. Reduce the transmission time $T_0^{(k+1)}(D)$ for retransmissions: this can be done by reducing the number of packets to be sent on retransmission. The negative acknowledgement can carry information as to which packets were successfully received.

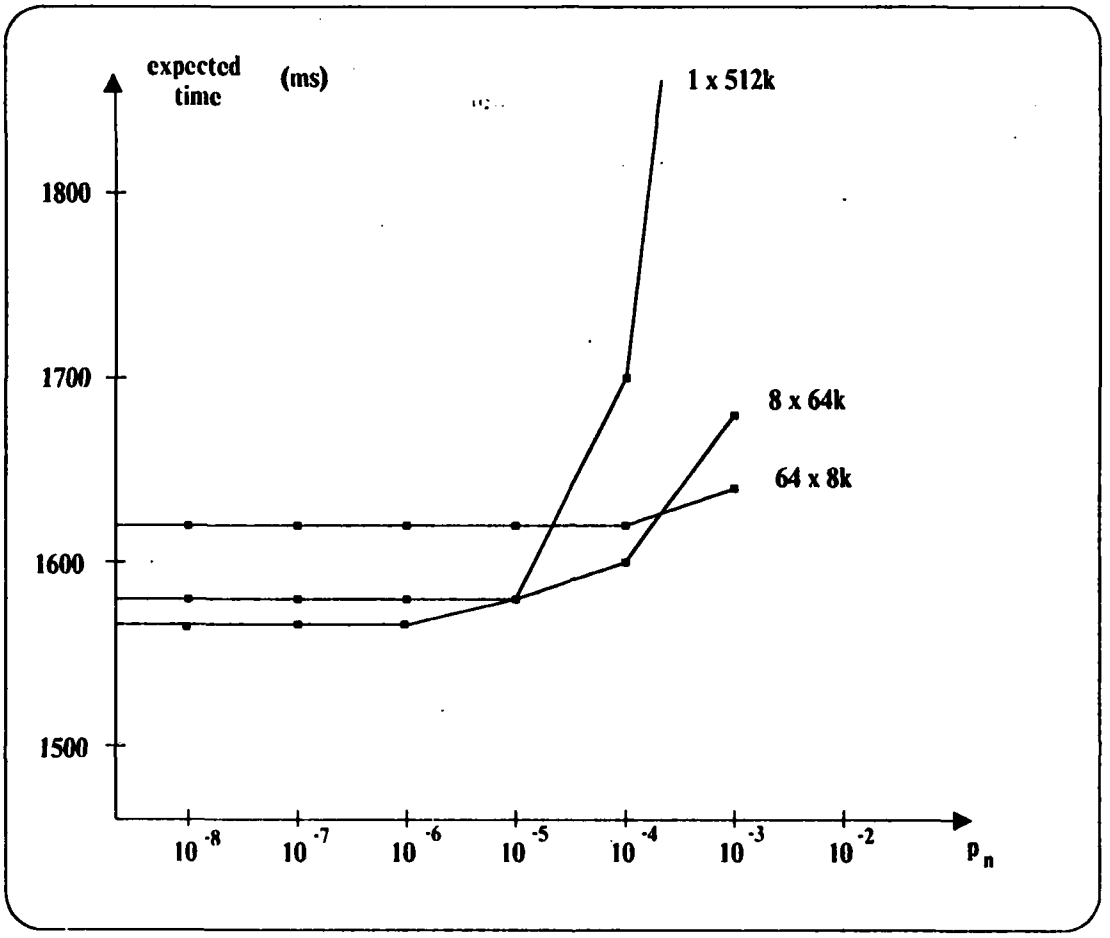


Figure 5-4: Expected Time for 512 kilobyte Transfer

3. Reduce the probability of failure of the retransmissions: since we are assuming independent failures, this probability is only dependent on the number of packets transmitted. Thus, here also reducing the number of packets sent has a beneficial effect.

Clearly, a combination of these different approaches seems optimal and to some extent straightforward. However, we analyze the different methods in isolation to assess their relative benefits. In particular, we consider the following retransmission strategies, starting from the most straightforward:

1. Full retransmission on error without negative acknowledgement (with different retransmission intervals).

2. Full retransmission on error with a negative acknowledgement after the last packet.

3. Selective retransmission.

In the case of full retransmission on error without negative acknowledgement, the probabilities q(i+1) form a geometric distribution with parameter

$$q = 1 - (1 - p_n)^{D+1}$$

Furthermore, for all k,

$$T_0^{(k+1)}(D) = T_0(D)$$
$$T_r^{(k+1)}(D) = T_r$$

The standard deviation of the elapsed time is easily shown to be

$$\sigma = (T_0(D) + T_r) \times (q^{1/2} \div (1-q))$$

This function is set out against p_n for different values of T_r in Figure 5-5. It is clear from the above formula and from the figure that the value of T_r has a significant effect on σ even for low error rates.

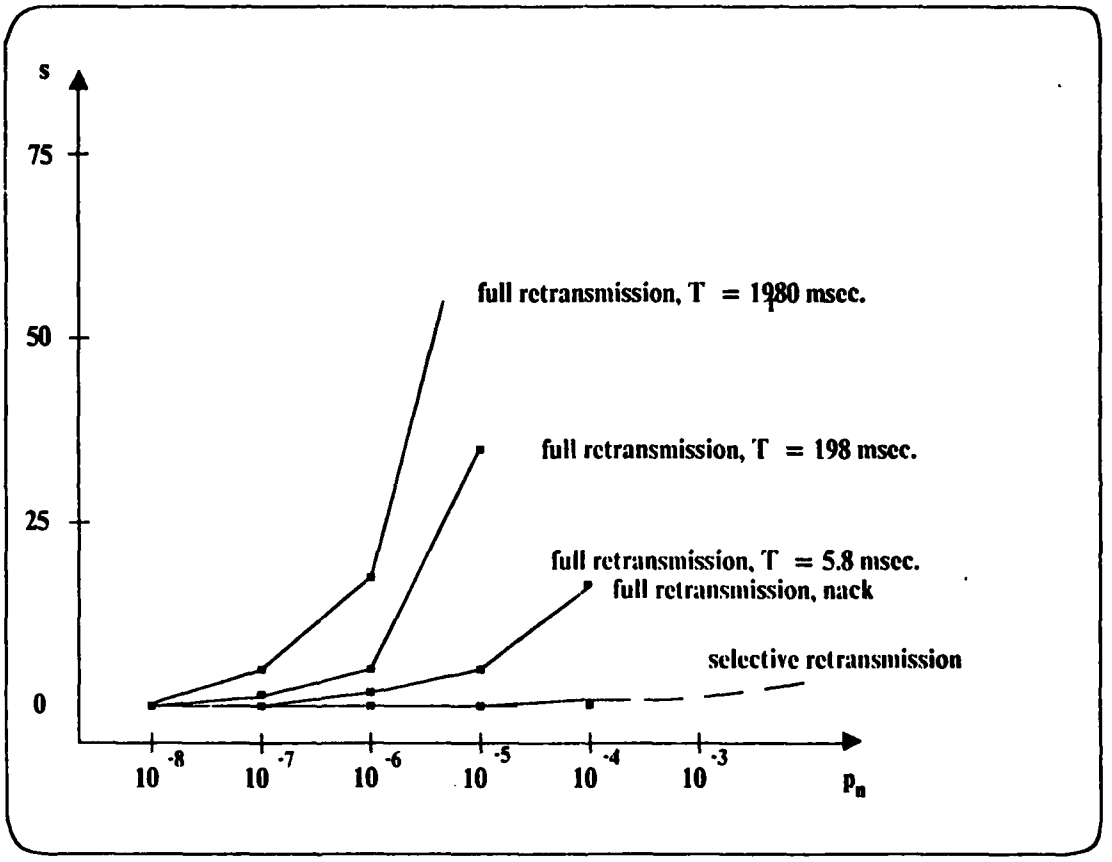


Figure 5-5: Standard Deviation for a 64 Kilobyte ***MoveTo***

Essentially, in order to achieve a low variance when using full retransmission, we need to choose T_r small compared to $T_0(D)$. This can be done as shown above by choosing a (physically) small value of T_r. Alternatively, an effectively small value of T_r can be achieved by using a negative acknowledgement, while still maintaining a much larger physical value of T_r. We use the following strategy:

1. If the recipient sees the last packet, it sends either a positive or a negative acknowledgement depending on whether or not it received all packets in the sequence.

2. If the sender gets a negative acknowledgement, or if the sender does not receive any acknowledgement within a time interval T_r, it retransmits the whole sequence of packets.

The characteristics of this strategy can be derived by the following argument. In the absence of a negative

acknowledgement, every failed transmission always takes $T_0(D)+T_r$. In the presence of a negative acknowledgement, the length of a failed transmission varies depending on whether such a negative acknowledgement was sent and received. Denote by $T_{f,k}(D)$ the length of the k-th failed transmission, and by $q(i+1)$ the probability of success on the i-th retransmission, then the expected time becomes

$$\mu = T_0(D) + \Sigma_{i=0}^{i=\infty} (\Sigma_{k=1}^{k=i} T_{f,k}(D)) \times q(i+1))$$

We approximate the inner sum by

$$\Sigma_{k=1}^{k=i} T_{f,k}(D) \approx i \times T_f(D)$$

where $T_f(D)$ is the expected time of a *failed* transmission[26]. Then, the expected time for the ***MoveTo*** becomes

$$\mu \approx T_0(D) + T_f(D) \times \Sigma_{i=0}^{i=\infty} (i \times q(i+1))$$

Noting again that

$$q(i+1) = q^i \times (1 - q)$$

with

$$q = 1 - (1 - p_n)^{D+1}$$

we get finally for the expected time

$$\mu \approx T_0(D) + T_f(D) \times (q \div (1-q))$$

For the values of p_n that are of interest, the expected time is approximately equal to $T_0(D)$. Given this, the variance is approximately

$$\sigma^2 \approx \Sigma_{i=0}^{i=\infty} (\Sigma_{k=1}^{k=i} T_{f,k}(D))^2 \times q(i+1))$$

Using the same approximation for the inner sum, and substituting for $q(i+1)$, we get

$$\sigma \approx T_f(D) \times (q^{1/2} \times (1+q)^{1/2} \div (1-q))$$

We now derive the value of $T_f(D)$. A failed transmission either takes $T_0(D)$ or $T_0(D)+T_r$. The first case occurs under the following conditions: not all of the (D-1) first packets arrived at their destination, the last packet arrived at its destination, and the negative acknowledgement arrived at its destination. The (conditional) probability of this happening, assuming the overall transmission failed, is the product of the individual probabilities of the above events, divided by the probability of a failure

$$[(1 - (1 - p_n)^{D-1}) \times (1 - p_n)^2] \div (1 - (1 - p_n)^{D+1})$$

For values of $p_n \ll (1 + D)$, this is approximately

$$(D-1) \div (D+1)$$

The failed transmission takes $T_0(D)+T_r$ in the case that either the last packet or the negative acknowledgement get lost. The probability of this happening, assuming there is a failure is

$$(1 - (1 - p_n)^2) \div (1 - (1 - p_n)^{D+1})$$

which, again for $p_n \ll (1 + D)$ reduces to

$$2 \div (D + 1)$$

The expected time for a failed transmission is then approximately equal to

$$T_f(D) \approx ((D-1) \div (D+1)) \times T_0(D) + (2 \div (D+1)) \times (T_0(D) + T_r)$$

If $D \ll 1$, then clearly

---

[26] This approximation becomes exact with probability 1 as i tends to infinity. For finite values of i, we believe it is a good approximation, since the values of $T_{f,k}(D)$ are densely clustered around their expected time $T_f(D)$.

$$T_f(D) \approx T_0(D)$$

and finally we get for the standard deviation

$$\sigma \approx T_0(D) \times (q^{1/2} \times (1+q)^{1/2} \div (1-q))$$

This formula indicates that the standard deviation when using full retransmission with a negative acknowledgement is all but independent of the retransmission interval. The values of σ for different values of p_n are set out in Figure 5-5 for comparison with full retransmission without negative acknowledgement.

By either choosing a small retransmission interval or by using negative acknowledgements, we have minimized the component of the standard deviation that is dependent on the retransmission interval. The standard deviation is also dependent on the amount of data retransmitted during retransmissions. This component can be minimized using selective retransmission.

Selective retransmission is implemented as follows. In order to execute a ***MoveTo*** containing D packets, (D-1) packets are transmitted without acknowledgement. The last packet is sent reliably: it is retransmitted periodically until an acknowledgement is received[27]. The acknowledgement to the last packet indicates which of the D-1 unreliably transmitted packets got to their destination. If D' did not get there, they need to be retransmitted using the same method: transmit D'-1 packets unreliably and the last packet reliably. This procedure continues until all packets get to their destination. The above observations allow us to derive the following recurrence relation for T(D). We denote by $T_{unrel}(i)$ the time necessary to transmit i packets unreliably. The probability that the full transmission succeeds on the first try is equal to

$$(1 - p_n)^{D-1}$$

The probability that i packets get lost on the first try is

$$C(D-1,i) \times p_n^{\,i} \times (1 - p_n)^{D-1-i}$$

where C(D-1,i) is the number of combinations of D-1 by i. The time T(D) to transmit D packets with selective retransmission is then

$$T(D) = (T_{unrel}(D-1) + T_{rel}(1)) \times (1 - p_n)^{D-1}$$
$$+ \Sigma_{i=0}^{i=D-1} [T(i) \times C(D-1,i) \times p_n^{\,i} \times (1-p_n)^{D-1-i}]$$

whereby furthermore

$$T(1) = T_{rel}(1) = T_0(1) + T_r \times (q \div (1-q))$$
$$q = 1 - (1 - p_n)^2$$

and

$$T_{unrel}(i) = i \times T_{unrel}(1)$$

The standard deviation associated with this retransmission strategy is difficult to derive analytically. Therefore we have simulated the procedure by computer and determined both the expected time and the variance from the simulation. Figure 5-5 shows the standard deviation observed in the simulation. The figure clearly indicates that the behavior of this retransmission strategy with respect to the standard deviation is superior to strategies that require full retransmission on error.

5.6.2.6 Conclusions

Given sufficiently low error rates, the expected time for large data transfers is almost solely dependent on T_0, the transfer time when there are no errors. Therefore, the retransmission strategy is of little or no influence on the expected transfer time. Assuming independent network failures, we have quantified the assumption of "significantly low error rates". We expect most local networks to perform according to this

[27] Although it is not shown here, this strategy performs identically to the previous one, if full rather than selective retransmission is done.

assumption. The advantages of streaming, i.e. transferring the data without per-packet acknowledgements, under these circumstances are quite clear.

When considering higher error rates, or more importantly, when considering the standard deviation of the transfer time at low error rates, the effects of different retransmission strategies start showing. In particular, we have shown that the standard deviation at low error rates increases with the amount of data retransmitted, the retransmission interval and the probability of further failures during retransmission (which is a function of the amount of data retransmitted, at least when assuming independent failures). A short effective retransmission interval can be achieved by returning a negative acknowledgement. Given the presence of such a negative acknowledgement, the amount of data can then be reduced by having the negative acknowledgement carry information about the packets received and by using selective retransmission on the part of the sender.

5.6.3. Remote Binding

A client can locate a server (i.e. find out his process identifier) by means of a ***GetPid***, assuming the server has previously done a ***SetPid***. A very brief discussion of the mechanisms supporting ***GetPid*** and ***SetPid*** is included in this chapter for the sake of completeness. An alternative implementation relying on the application-level use of multicast, is described in Section 6.6.2. This implementation does not require any kernel or protocol support whatsoever and results in a far more flexible binding mechanism.

In order to support the ***SetPid-GetPid*** mechanism, each kernel maintains a table, called the ***LogicalIdMap***. This table has a configurable number of entries, each of which has two fields, namely a process identifier and a scope. In response to a ***SetPid(logicalId,pid,scope)***, the kernel fills in the corresponding entries of the table entry with index ***logicalId*** with the given ***pid*** and ***scope***. Any previous entry is overwritten.

GetPids are handled in the following way. First, when the scope of the ***GetPid*** is ***local***, the kernel simply looks up the table entry with index ***logicalId*** and returns the corresponding process identifier, assuming the scope of the entry is not ***remote***. When the scope is ***any***, then the local table is searched first, and when no successful match is found, or when the process identifier matching the index is found to correspond to a no longer existing process, the remote ***GetPid*** routine is invoked. The latter routine is invoked immediately if the scope of the request is ***remote***.

The remote ***GetPid*** broadcasts a ***remoteGetPid*** packet on the network asking all kernels if they have the desired mapping[28]. When such a ***remoteGetPid*** packet arrives, each kernel looks in its ***LogicalIdMap*** and checks whether it has a process registered for the requested logical identifier (with scope either ***any*** or ***remote***). If so, and if the process identifier registered corresponds to a valid process, it puts a ***remoteGetPidReply*** packet on the network. This packet is sent point-to-point to the requesting kernel. Many such packets may arrive at the requesting kernel, since many machines may have a process registered for that logical identifier. The requesting kernel simply takes the first to arrive.

This concludes the specification of the V interkernel protocol and the discussion of its characteristics. We now turn our attention to some aspects of the implementation of this protocol as part of the distributed V kernel.

5.7. Some Aspects of the V Kernel Implementation

The V kernel has three main modules: the interprocess communication module, the device module, and the process and memory management module. The way these are structured has some interesting repercussions on the way remote kernel operations can be performed. This is discussed in Section 5.7.1. We next turn our attention to the interprocess communication module and in particular to the way in which it accommodates remote interprocess communication. We single out two topics for discussion. First, network interprocess

[28] The logical identifier is contained in the ***forwarderPid*** field.

communication in the V kernel is handled within the kernel (as opposed to through a process-level network server). We discuss the motivation for this choice and its ramifications. Second, we show how by proper design different kinds of network interfaces can be accessed through a common procedural interface, while still taking advantage of the capabilities of each interface.

5.7.1. Overall Kernel Structure

The kernel consists of three main modules: the interprocess communication module, the device server pseudo-process, and the kernel server pseudo-process. The latter two are called *pseudo-processes* because, although to other processes they look like regular processes (and the normal interprocess communication primitives are used to communicate with them), in reality they are a set of procedures within the kernel. The *kernel server* is a pseudo-process that provides a message interface to the kernel's process and memory management operations. The *device server* provides message-based access to devices. In practice, clients communicate with the device server more conveniently through the V I/O protocol [13]. The kernel commonly supports the keyboard, the console, the frame buffer, the mouse, the Ethernet and the clock. The kernel can also be configured to support one or more disks, for the file server or for workstations with local disks.

Both from an efficiency and from an integrity standpoint, it is attractive to perform process, memory and device management in the kernel. For instance, some of the kernel server operations manipulate the kernel's process descriptor data structures. From an integrity standpoint, it is desirable that these data structures are only accessible from inside the kernel. Similarly, having the kernel manage devices allows some measure of protection since the actual device operations are now executed by the kernel and not by a process whose authorization might be difficult to check. Second, access to devices usually requires execution of privileged instructions or access to protected memory locations, so part of the device operation has to be done in privileged (kernel) mode anyways. Doing device management outside the kernel, while saving on kernel space and complexity, therefore usually requires two extra context switches and additional copying of data. This can significantly degrade overall performance [81]. Note that, unlike in Unix 4.2 BSD [52] for instance, the file system and the internet protocol software still reside outside of the kernel. Only the device drivers for the disk and the network interface are present in the kernel. The resulting increase in kernel size is relatively small and well justified given its performance benefits.

Given the fact that we want to implement these functions in the kernel, we observe a number of significant advantages to doing so by means of pseudo-processes which are accessed by the regular message primitives [63]. First, it provides a uniform model of access to clients, allowing them to be oblivious to the fact that these operations are actually implemented in the kernel. Second, there is no additional protocol complexity for implementing these kernel operations remotely. For instance, destroying a remote process (subject to certain privileges) is done by sending a message to the remote kernel server. The V protocol can be used to carry this message without any need for additional protocol primitives (packet types). Finally, this design has a number of potential advantages that have at this point not fully been explored: it reduces the number of kernel primitives making it possible to have a single machine "trap" per primitive; it effectively separates the implementation of interprocess communication from other kernel services, allowing one to contemplate hardware or microcode support for the interprocess communication module; and finally, it allows for interposition of a debugger or monitor process.

Although part of the general interprocess communication, device access is not transparent. It is quite possible to remedy this situation, but this has not been done so far for apparent lack of demand and because of the costs involved. These costs result primarily from the need to buffer large amounts of data in the kernel. This would become necessary if data from a device were to be sent to a remote process. Indeed, that data would have to be kept around for potential retransmissions. Similarly, some of the kernel server operations are restricted to privileged system processes. For instance, allocating an address space for a new program to run in, is restricted to the so called *team server* process. In order to start a program remotely, one therefore has to go through the team server on the remote machine, which can be instructed whether or not to accept such remote requests. We have found this restriction to be useful since the kernel server operations manipulate rather vital machine resources.

5.7.2. The Interprocess Communication Module

When a kernel request is directed at a remote process, or when a packet carrying a remote request comes in, the network interprocess communication module is invoked. In the V kernel, the entire network interprocess communication module resides within the kernel. For instance, for a ***Send*** addressed to remote process, we call a ***NonLocalSend*** routine in the kernel, which proceeds to put a packet on the network. When that packet arrives at the receiver's kernel, the network interrupt handler invokes the appropriate kernel routine to process the message, i.e. put it in the receiver's message queue and if necessary, unblock the receiver.

An alternative organization would be to have a process-level *network server* handle the network aspects of remote request. Again taking the example of a ***Send*** to a remote process, the kernel now forwards the request to the network server process, which (most likely via another kernel call) handles transmission of the packet over the network. On the receiving end, the kernel network device handler passes the packet on to the network server on his machine, which then forwards it, again via a kernel call, to the final receiver.

5.7.2.1 Performance Issues

The tradeoff between the two approaches is mainly one of performance. Implementing network interprocess communication outside of the kernel causes extra kernel traps, extra process switches and extra copy operations. Let us follow in detail the execution of a remote ***Send-Receive-Reply*** sequence under both implementation strategies and under the assumption that no other processes than the sender, the receiver, and the respective network servers are present. Consider Figure 5-6. In this figure, an arrow represents a transfer of control in or out of the kernel. For the purposes of interprocess communication each transfer causes a copy of the message to happen. We count the expense of the different operations as follows: a kernel trap is the sum of an entry into and an exit from the kernel; a process switch is the loading of the volatile storage associated with a process. From Figure 5-6 it can be seen that an implementation of remote interprocess communication in the kernel requires 2 kernel traps, 0 process switches and 4 copies. An implementation outside of the kernel requires 6 kernel traps, 4 process switches and 12 copies. A kernel trap requires approximately 60 microseconds, a process switch about 150 microseconds and copying 32 bytes takes approximately 25 microseconds. This would indicate that an implementation outside of the kernel would require approximately 1 millisecond more in processor time on the sender and the receiver machine combined. The difference in elapsed time is more difficult to estimate given the potential for concurrency. Besides, extra machinery for generating timeouts and retransmissions causes additional process switches, and thus additional expense. Moreover, as has been noted in many process-level protocol implementations [20], process scheduling can be a additional source of real-time delay.

5.7.2.2 Synchronization and Interrupt Disable Time

Clearly, then, there are significant performance gains to be obtained from a kernel-level implementation of the remote interprocess communication module. However, such a concession with respect to modularity is not without its price. The handling of a remote request is now done at the network interrupt level. Since handling such a request may involve manipulating message queues, it needs to be synchronized with local (kernel) operations manipulating the same data structures. Additionally, the timer interrupt can cause various retransmissions or timeouts to take place, again potentially accessing message queues. We first show how we maintain synchronization by selectively turning off interrupts. We then provide some estimates with respect to maximum interrupt disable time.

Although a finer grain of synchronization is conceivable, we have chosen to turn off all network interrupts during kernel operations (both for local and remote requests). Additionally, with respect to remote operations, a timer interrupt can cause one of three things to happen:

1. A request for a remote process is retransmitted.
2. A request for a remote process is timed out, and the requester is added to the ready queue with some appropriate indication of failure.
3. A request from a remote process is timed out (i.e. the kernel has not received any retransmission of the

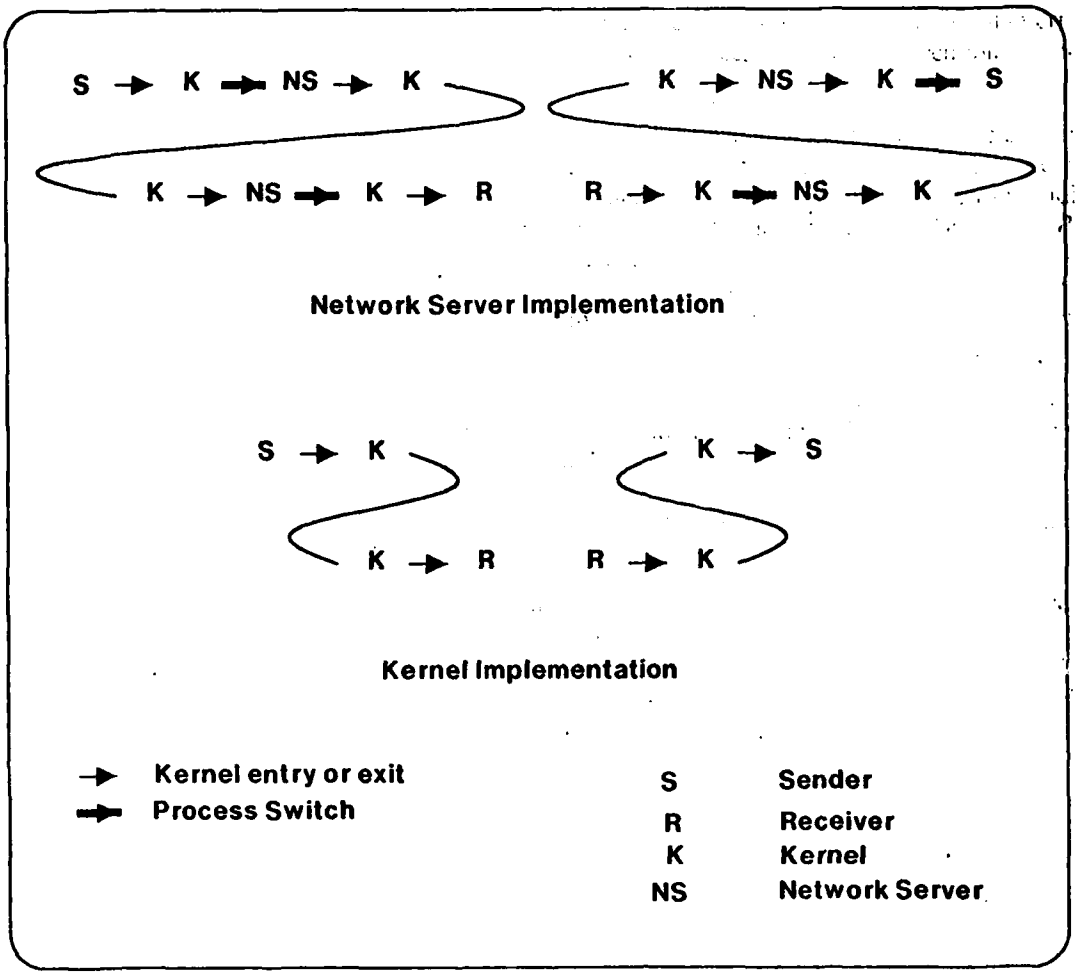


Figure 5-6: Kernel-level vs. Process-level Implementation

request for a time longer than the timeout interval T_t), its packet buffer is deallocated and references to it are removed (Most likely, such a reference would be in the message queue of a local process.)

Turning off the timer interrupt altogether during kernel operation would distort the proper operation of the kernel's timing services. Nevertheless, we must prevent uncontrolled concurrent access to message queues. Therefore, we allow the timer interrupt to occur at any time, and we allow the corresponding routine to perform all of its duties, except when the timer interrupts while the kernel is running. In that case, we prevent it from accessing the message queues.

Having thus achieved synchronization by selectively turning off interrupts, we would like to convince ourselves that the maximum interrupt disable time remains relatively small. This is especially important for the timer interrupt since prolonged disabling might mask the next timer interrupt. Reviewing the various events a timer interrupt might cause, the most common operation is the (re)transmission of a single packet. This causes the timer interrupt to remain disabled while the packet is being copied out from main memory to the network interface (The actual network transmission proceeds asynchronously.) For a maximum size packet, we can estimate this copy to take about 1.5 milliseconds, significantly lower than the timer interval of 10 milliseconds. The situation could degrade significantly if network transmission had to be done

synchronously on a slow network or if the timer interval was chosen much shorter. A very unlikely worst-case can occur as a result of a remote request being timed out. This causes the kernel to search through the message queue of the intended recipient of that request in order to remove the message buffer from that queue. While potentially long, the queue length and thus the search time are still bounded by the maximum number of message buffers the kernel has allocated.

Network interrupts are disabled altogether during kernel operation. Prolonged disabling of the network interrupts could cause a problem if packets are not removed quickly enough from the device input queue, causing the queue to overflow and packets to be dropped. While this is not in contradiction with the definition of the Ethernet data link level, it could lead to undue performance degradation. We have no data available on the percentage of time or the maximum length of time the network interrupts are disabled. In practice, it has not been observed to be a problem.

5.7.2.3 Concluding Remarks

Placing network interprocess communication inside or outside of the kernel is a tradeoff of performance vs. modularity. We have quantified the cost associated with using a network server process for remote interprocess communication. We have also shown the problems one has to deal with as a result of a kernel-level implementation, particularly with respect to interrupt disabling. Our experience indicates that when proper care is taken in the implementation, and when high performance interprocess communication is desired, the performance advantages of a kernel-level implementation outweigh the disadvantages.

5.7.3. Packet Transmission and Reception

At present, we are using or anticipating to use a number of network interfaces with quite different characteristics. We divide those interfaces into four categories according to their capabilities.

1. **Programmed I/O interfaces:** the processor has to copy the packet from main memory over the I/O bus to the network interface. The network interface is assumed to generate a single interrupt on receipt or on successful transmission of a packet.

2. **DMA interfaces:** the network interface is capable of accessing a contiguous DMA buffer in main memory.

3. **Scatter-gather DMA interfaces:** the network interface is capable of composing a packet out of a number of discontiguous buffers in main memory.

4. **Programmable network interfaces:** the network interface can be programmed on-board; it uses DMA to access data in main memory.

In implementing remote interprocess communication, we would like to provide a single, device-independent procedural interface to these different devices. Nevertheless, we want this procedural interface to allow the implementer sufficient freedom to take advantage of the capabilities of a specific interface. In particular, we would like to avoid all unnecessary copying.

5.7.3.1 Packet Transmission

When transmitting a packet, (most of) the header and the message are present in the process descriptor of the sending process. The data segment, if any, is in general in the process' address space. In order to take maximum advantage of the capabilities of various network interfaces, we use the following technique. The process descriptor is laid out in such a way that the part of it that contains the packet header information and the message is laid out contiguously, in the same format as the packet itself. This requires only a minor amount of extra space in the process descriptor (20 bytes). Three extra fields are allocated in the process descriptor for the purpose of remote communication: ***segmentPointer*** and ***segmentSize*** indicate the beginning and the size of the segment, if any, that is associated with this request. ***currentPointer*** is reserved for use by the network interface (see below). The segment size is not restricted to a single packet: for ***MoveTo*** and ***MoveFrom*** operations, it consists of the entire segment to be transmitted. In order to have the packets

associated with this request transmitted onto the network, a single call is made on the routine

```
WriteKernelPacket(pd)
```

where ***pd*** is a pointer to the process descriptor of the requesting process. Although the procedural interface to ***WriteKernelPacket*** is independent of any particular network interface, the further execution of the call depends entirely on the particular interface used.

On a programmed I/O interface, the processor moves the packet header from the process descriptor into the network interface, followed by the first part of the segment that fits in a packet. If more than one packet needs to be sent, the process descriptor is put on a queue. When the transmitter interrupt indicates transmission of the first packet is finished, the next packet is sent in the same way. The ***WriteKernelPacket*** routine uses the ***currentPointer*** field in the process descriptor to keep track of where the data segment of the next packet is to come from. For the purpose of speeding up ***MoveTo*** and ***MoveFrom***, it is extremely attractive to have a network interface that provides double buffering on transmission. Referring back to Figure 3-8, this would allow us to operate the network at a speed close to its advertised data rate.

For a simple DMA interface, the header and the appropriate part of the segment are first copied into a contiguous location in (kernel) memory before the DMA operation is set up. For remote V interprocess communication, there are no advantages to using such an interface, since the processor must make a copy of the packet, as for programmed I/O interfaces. Scatter-gather DMA devices avoid this copy by passing the DMA device pointers to the header and the appropriate part of the segment. Multi-packet transfers are done in the same way as for programmed I/O interfaces.

If the network interface can be programmed on-board, we would like the procedure ***WriteKernelPacket*** to execute entirely on the interface. This would not only relieve the processor of packet transmissions but also of the management of the queue for multi-packet transfers. The processor would then only be interrupted when the whole packet sequence has been transmitted. Additionally, depending on the sophistication of the board, one could also offload retransmission and timeout management to the interface.

5.7.3.2 Packet Reception

On the receiving end, it is more difficult to provide a uniform interface to the different network devices. In order to reduce the amount of specialized code associated with remote interprocess communication, remote messages are queued in *alien* process descriptors: these are regular process descriptors but they represent messages from remote process. The kernel always has an alien process descriptor ready for incoming requests.

On a programmed I/O device, the first few words of the packet are read into the packet buffer, just enough to read the protocol type. This field allows the kernel to distinguish interkernel packets from regular Ethernet access. If the packet is an interkernel packet, the rest of the packet header is read into the packet buffer. If a segment is appended to the packet, it is moved immediately into its final destination.

Regardless of whether one uses a simple or a scatter-gather DMA interface, there does not seem to be a simple way in which to avoid an extra copy if a segment is appended to an incoming packet. For simple DMA devices, the entire packet has to be read into a contiguous DMA buffer. We let the initial part of this buffer correspond to the process descriptor for the incoming request. The segment is then copied by the processor to its final destination. The same holds essentially for scatter-gather DMA interfaces: the extra copy is necessary because the kernel has to look at the header before it can decide where the segment goes. One could however avoid the copy by receiving the segment in a buffer aligned on a page boundary. If the corresponding client buffer is page-aligned as well, the page could then be mapped from the kernel's address space into the client's without an extra copy.

Finally, if the interface can be programmed on-board, the program executing on the board could look at the header and DMA the appended segment immediately to its final location. Additionally, such an interface could provide a number of useful functions, such as discarding broadcast packets not of interest to this kernel, etc. We believe such an interface could significantly offload the processor of its duties with respect to network interprocess communication.

5.8. Chapter Summary

In this chapter we have related to the reader some important experience gained from the design and the implementation of the V interkernel protocol. We have separated the definition of the protocol from its specific implementation in the V kernel. The definition of the protocol consists of a set of rules governing process naming, process location, packet format and packet sequencing. In summary, we have shown that

1. Domain-wide unique identifier generation by announcement-complaint algorithms can be made arbitrarily failure-resistant at the expense of increased running time of the algorithm.

2. Although in a broadcast domain, process location by broadcasting is the method of choice, we have chosen point-to-point communication to reduce the overhead on unrelated processors.

3. With respect to message exchanges, we have quantified the cost of exactly-once semantics and explored the potential of idempotent operations.

4. The expected time of large data transfers is primarily dependent on the no-error transfer time which must therefore be optimized by streaming. The variance is dependent on the retransmission strategy: negative acknowledgements and selective retransmission are suggested.

This protocol has been implemented in the distributed V kernel and has been in use as such for almost two years. Among the implementation aspects, we have singled out two important aspects for discussion:

1. We have shown that the problems of a kernel-level implementation of network interprocess communication can be avoided if appropriate care is taken in the implementation, and that the performance benefits in our environment are significant.

2. We have explored the potential of offloading network interprocess communication to intelligent, programmable network interfaces.

Many of the design decisions are predicated on intuitive notions about the underlying hardware and about the usage of the protocol. In this chapter we have developed a set of expressions relating the performance of the protocol to characteristics of the hardware and to a lesser extent, usage characteristics. While we have some idea of how the hardware behaves, no data of any significance is available on usage patterns in a distributed system. Whether our decisions are well matched with typical usage patterns therefore remains an open question.

—6—
One-to-Many Interprocess Communication

6.1. Introduction

So far we have discussed the most common form of message-based interprocess communication, namely *one-to-one communication:* a single sender sends a message to a single receiver and (usually) gets a response back. For example, a process reads a file page by sending a message to the file server requesting that particular page. The file server then returns the page in a reply message. However, a process may need to communicate with a group of processes to either *notify* them of some information or to *query* them for information. For example, a process needing file service may want to query all server processes to determine which one is a file server, or which server stores the desired file.

The need for one-to-many communication becomes apparent in distributed systems, where the processing and storage facilities are divided among several nodes connected by a network. In this environment, in the absence of global shared memory, one-to-many communication is used to locate services that would otherwise be advertised in a table stored in global memory. In general, shared memory can be viewed as a broadcast communication channel (with the additional capability of storage). Information of general interest is posted in shared memory and then searched to satisfy queries from different processes. Broadcast provides a similar communication function, but without storage. Information is multicast to a set of possibly interested parties or, alternatively, a process multicasts to a set of relevant parties to locate the desired information.

Fortunately, most distributed systems are connected by local network and bus technologies that provide efficient multicast communication, i.e. one-to-many *interhost* communication. For example, in shared bus technologies like Ethernet [27], each packet is essentially received by all hosts and only a filtering mechanism based on destination address gives the appearance of point-to-point communication. By providing a filter that accepts multiple specified addresses, (unreliable) multicast is available at no extra cost over one-to-one communication, both to the sender and all of the receivers. Thus, to communicate without guaranteed delivery with N workstations takes a single packet on a multicast network whereas simulating this using point-to-point communication takes N packets. Clearly, an application needing one-to-many communication (which may be unreliable) can be implemented much more efficiently when provided with efficient access to local network multicast than when forced to use one-to-one communication for the same purpose.

This chapter describes an extension of the V interprocess communication to provide one-to-many communication to applications. An earlier version of this chapter appeared in [18]. Section 6.2 describes the considerations taken into account when integrating multicast. Section 6.3 describes the model of one-to-many communication we support and the extensions to the kernel interface necessary to provide this facility. Section 6.4 presents motivations for our design choices. Section 6.5 discusses the implementation. Experience with applications and ideas for use are presented in Section 6.6. Section 6.7 describes related work. Conclusions are drawn in Section 6.8 in which we also point out some remaining open questions.

6.2. One-to-Many Communication Considerations

Several considerations are recognized in extending the V kernel to provide one-to-many communication. First, one-to-many communication should be provided as an integral, transparent part of one-to-one interprocess communication. Processes should be able to send, receive and reply to a message addressed to multiple processes in the same way as if the message were addressed to a single process. This avoids redundant code in the kernel and awkward programming at the process-level. We do not however preclude the ability for more sophisticated applications to discriminate group messages when necessary.

Second, we are interested in determining whether the V kernel's interprocess communication is smoothly extensible to one-to-many communication. The large body of existing software that uses the one-to-one primitives and the favorable experience in using it make it unattractive to make unnecessary changes to it or to develop an independent design. However, we do not place compatibility with the existing kernel interface as a unquestionable constraint on our thinking.

Third, one-to-many communication should be efficient in terms of delay, network bandwidth and processor usage, both relative to the performance provided by hardware multicast as well as relative to the performance of one-to-one communication. We are particularly interested in using one-to-many communication to structure the communication aspect of highly parallel distributed computation on a set of workstations with no shared memory (See Section 6.6.3). Efficient communication is crucial in achieving this goal.

Finally, in keeping with the design philosophy of the V kernel, we wish to have the kernel provide a simple but efficient and flexible mechanism such that more powerful constructs can be built at the application level (or run-time support level).

6.3. One-to-Many Extensions

6.3.1. Communication Model

A process *group* is a set of one or more processes, possibly on different machines, identified by a *group identifier*. All processes in a group are equal; there are no distinguished members. Processes can freely join or leave groups, and are free to join multiple groups.

A group identifier is identical in format to a process identifier. Sending to a group involves specifying a group identifier to ***Send*** instead of a process identifier. Any process can send to a group, including non-members.

A process may elect to receive zero, one or multiple reply messages in response to a message sent to a group. The kernel handles each case as follows. The zero-reply case is handled as an unreliable multicast to the group with the sending process not blocking. The one-reply case blocks the sender to receive one reply message, assuring reliable delivery to at least one process. Further replies are discarded and no indication is given as to how many other processes received the message or replied to it. This form of communication is the same as one-to-one reliable communication to the first respondent and unreliable datagram communication to the rest of the group.

The multiple reply case is similar to the one-reply case except that the second and subsequent reply messages are queued in the kernel for the sender to retrieve, up until the start of the next message transaction, i.e. the next ***Send***. It is left to the sending process to decide how many replies it wishes to receive and how long it is willing to wait for them.

The addition of no-reply and multiple-reply ***Send*** presents somewhat of a departure from the synchronous message-passing present in Thoth and carried forward in the one-to-one communication in V. This departure is necessary in order to accommodate certain useful applications of one-to-many communication. The model of one-to-many communication is implemented by adding some new primitives to the kernel and by extending the semantics of some existing primitives.

6.3.2. New Primitives

groupId = AllocateGroupId()

Allocates and returns a group identifier with the invoking process becoming a member of the new group. Group identifiers are identical in syntax and similar in semantics to process identifiers.

JoinGroup(groupId, processId)

Makes the process with process identifier ***processId*** a member of the group with group identifier ***groupId***. In particular, all messages sent to ***groupId*** are received by this process (with some degree of reliability). The kernel makes no effort to ensure that ***groupId*** represents an existing group.

LeaveGroup(groupId, processId)

Removes the process with process identifier ***processId*** from the group with group identifier ***groupId***.

processId = GetReply(replyMessage)

Returns the next reply message from a group ***Send*** in ***replyMessage*** and the identity of the replying process in ***processId***. If no reply message is available, ***GetReply*** returns with ***processId*** set to 0. Additional reply messages for this particular transaction may be received and, if any, will be returned on a later invocation of ***GetReply***. It is thus left to the sender to decide on what basis to determine that it has received enough replies. However, all replies for a particular message transaction are discarded when the process sends again, thus starting a new message transaction.

There is a range of (statically allocated) reserved group identifiers, disjoint from those allocated by ***AllocateGroupId()***. The use of these group identifiers is similar to the *well-known sockets* in PUP [11] and other protocol families.

6.3.3. Changes to Existing Primitives

pid = Send(message, groupId)

Sends a message to all processes that are a member of the group with group identifier ***groupId***.

Two flag bits are reserved in the message to indicate whether the process expects no reply, a single reply or many replies. If no reply is expected, the ***Send*** returns immediately with ***groupId*** as return value. The reliability of this ***Send*** is equal to that of the underlying data link of the network. If a reply is expected, the process is blocked until the first reply arrives. If no reply arrives within a certain time interval, the request is retransmitted and eventually, after a number of retransmissions, timed out. Multiple replies are received using ***GetReply*** after the first one has been received. However, retransmissions stop after the first reply is received. Additional reply messages may be queued any time up until the end of the current message transaction, signaled the next ***Send*** from this process.

processId = Receive(message)

We assume the invoking process is a member of the group to which process ***processId*** has sent. Receive returns ***processId*** and the message exactly as if the sending process had sent directly to the receiver. The receiver is also expected to ***Reply*** to ***processId*** exactly as in the one-to-one case (rather than to ***Reply*** to the group).

6.3.4. Example

Figure 6-1 gives a detailed description of the exact sequence of events on a multiple-reply ***Send***. First, the process executes a ***Send*** to a group identifier and indicates in the message that it expects to receive more than one reply. The process gets blocked and the distributed kernel takes care of delivery of the message to the members of the group (with datagram reliability). As soon as the first reply arrives, the process gets unblocked with the reply message being returned from the ***Send***. Further replies are buffered in the kernel. If it wished to do so, the sender could stop receiving replies at this point. In this example however, the sender elects to look at further replies, and thus executes some number of ***GetReply*** calls. Finally, after it has seen the third reply message, the process decides not to receive any further replies. If, as is the case in this example, further replies come in, they will be discarded when the sender executes another ***Send***.

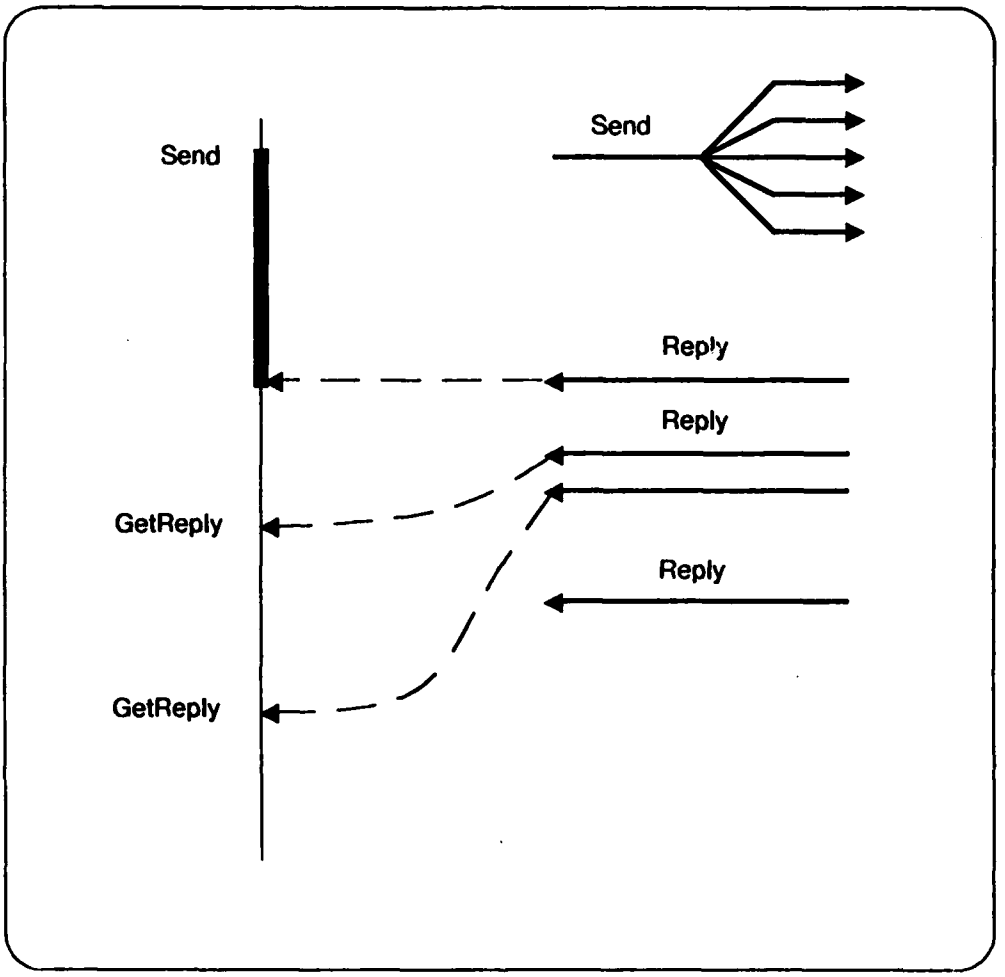


Figure 6-1: Multiple-Reply One-to-Many Communication

6.4. Design Rationale

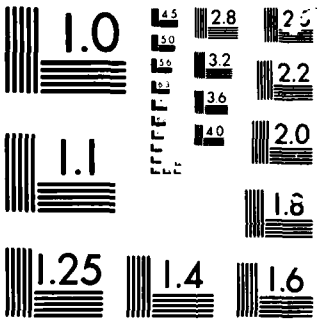
We first argue that some level of kernel support significantly enhances the ease and efficiency with which one-to-many communication can be used.

6.4.1. Kernel Support

Superficially, one could claim that one-to-many communication can be implemented using multiple one-to-one messages, and that no special kernel support is necessary. However, a number of problems arise with this approach. First, one may not know the identity of all the desired receivers of a message if, for example, the communication is analogous in intent to advertising. Second, when there is a large number of recipients, the cost of sending separate messages is significant, especially relative to that possible when the underlying hardware provides efficient broadcast or multicast. Finally, related to efficiency, some forms of one-to-many communication do not need the full reliability associated with one-to-one communication. This unneeded reliability, if inherited from the one-to-one communication primitives, imposes a further cost on

AD-A166 936 MESSAGE PASSING ON A LOCAL NETWORK(U) STANFORD UNIV CA 2/2
DEPT OF COMPUTER SCIENCE W ZWAENEPOEL OCT 85
STAN-CS-85-1083 MDA903-80-C-0102

UNCLASSIFIED F/G 17/2 NL



MICROCOPY CHART

communication. Adequate support for one-to-many communication can relieve all of these problems. Next, we survey potential uses of one-to-many communication in order to derive an appropriate level of kernel support.

6.4.2. Use of One-to-Many Communication

The two generic uses of one-to-many communication are *notification* and *query*. Notification consists of communicating specified information to some group of processes. Query involves extracting specified information from a group of processes or determining to some level of certainty that the desired information is not available from this group. Query can be replaced by a notification facility using the idea of *advertising*. For instance, a client wishing to locate a service may implement the query by notifying the group of server processes of its interest in a particular service. It then waits to receive messages from members of this group in response to its advertisement. An inverse approach used in the Port system [51] is for servers to advertise their identity and services. A client simply waits to receive an advertisement for the service of interest to it.

In the V-System, both notification and query are implemented by sending a message (to many processes). In the case of a notification, the replies to the message serve purely as acknowledgements. In the case of a query, the replies carry reply data as well as serving as acknowledgements.

In first approximation, notifications can be classified according to the number of reply messages (acknowledgements of receipt) that are required. Similarly, queries can be classified according to the number of responses that are expected to the query. We define *N-reliable* notification (query), as a notification (query), for which N replies are required. For notifications, N ranges from zero to the number of members in the group. For query, N ranges from one to the number of group members. *All-reliable* notification and query (where responses are expected from all members of the group) require that the membership of the group be known.

The classification in the above paragraph considers termination of a notification (query) depending on receipt of a specified number of replies by the sender. Closer observation reveals that it is useful to be able to terminate notifications (queries) when a more general *termination condition* is satisfied. Consider for instance the case where we want to execute a program remotely, preferably on a lightly loaded processor. Each machine runs a *team server* process, which maintains relevant information about the state of its machine: whether it is available for remote execution, what its current processor load is, etc. All team servers belong by default to the predefined *team server group*. In order to find a suitable host for remote execution, we send a query to the team server group, indicating that we wish to receive many replies. Several strategies are then possible. For instance, we could stop receiving replies when an idle processor has been found, or, failing that, after a certain time interval or after a number of replies have been received, at which point we would pick the processor with the lowest load. In this case, the termination condition is: either an idle processor has been found, or a certain time interval has expired or a certain number of replies have been received. Similarly, for notification, a more general termination condition might be, for instance, that an acknowledgement from a particular process has been received. Viewed from this angle, N-response query and notification are special cases where the termination condition states that N replies must be received.

We conclude from the above discussion that in general some facility is needed whereby a message is delivered to a group of processes, and replies are returned until some termination condition is satisfied. We have chosen to partition the implementation of this facility as follows: the kernel takes care of delivery of the message to the group members and returns the replies from any number of them. It is then left to the sender to evaluate the termination condition and to decide if further replies are required. This division of labor is based on the observation that it is clearly undesirable to require the kernel to decide some arbitrarily complex termination condition. In fact, one might even imagine cases where it would be near to impossible for the kernel to make such a decision, for instance when the termination condition depends on information the process might have obtained via shared memory while replies are still outstanding. Via the multiple-response option of group *Send*, the kernel supplies the sender the basic mechanism to be able to decide an arbitrarily complex condition. The one-reply option could be viewed as a special case of this general facility, the termination condition being that one reply is received. This special case is supported by the kernel for two

reasons:

1. It allows the simple case of a one-to-many ***Send*** with one reply to appear to the sender as completely identical to a one-to-one ***Send***. This is desirable for many simple applications.

2. The specification of a single reply allows an optimization in the kernel. Indeed, all replies after the first one can be discarded immediately. Additionally, this optimization can be performed without any extra mechanism in the kernel.

Another special case is all-reliable one-to-many ***Send***. This option can be built on top of the available kernel primitives (See Section 6.4.3 for a discussion of different implementation strategies). However, it is not supported by the kernel itself because the benefits of such kernel support do not seem to warrant the added complexity. The added complexity results from the need to know the membership of the group in order to implement all-reply one-to-many ***Send***. In general, the kernel on the sender's machine does not possess a record of the membership of any group. Maintaining such knowledge is not required for many applications, and would be cumbersome to implement for groups where processes can freely join, leave or be destroyed. For the sender to supply a list of member processes to the kernel would require adding another primitive to the kernel. This addition has been rejected, particularly in view of the fact that a kernel implementation of all-response ***Send*** would not result in better performance in terms of network or processor utilization, as is shown next.

6.4.3. Implementation of All-Replies Send

Several methods can be used to implement all-reliable one-to-many ***Send*** on top of the available kernel primitives. First, a process can use repeated group ***Sends*** until it receives replies from all members of the group. This method is not recommended because of its high cost in terms of the number *packet events*. A packet event is defined as the transmission or reception of a packet (See Section 5.3.2). The number of packet events is indicative of the processor time spent in communicating. Given that processor usage is a critical aspect of local network performance, a large number of packet events may degrade performance significantly. With N members in a group not including the sender (and all members on different machines), the above implementation requires one multicast packet and N point-to-point acknowledgement packet per transmission. The cost in packet events is $3N+1$ without errors and an extra $3N+1$ for each error. This contrasts with $4(N-1)$ packet events for N-1 point-to-point errorless messages and 4 packet events for each additional error. Thus, the multicast scheme is more expensive in packet events than the point-to-point simulation of multicast if there are errors and N is greater than 1.

Note that one might expect to lose packets in reply to a multicast packet because each of many reply packets could arrive simultaneously (back-to-back) at the sending host, causing it to fall behind and drop some of the packets. The solutions to this proposed by Mockapetris [55] are to use sophisticated *filter* network interfaces to offload the processor and to use two multicast addresses for one group, alternating between them on each packet so that retransmissions only appear on the previous address and are not seen twice. The former are not available and the latter only works if there is only one sending process, a restriction that is unworkable in our applications.

A more appropriate strategy is to ***Send*** the request to the group by one-to-many communication, collect the replies, and then retransmit the message one-to-one to those group members that did not (timely) reply to the original ***Send***. This avoids an excessively high cost in packet events. For instance, if two processes failed to respond, it would cost $3N + 9$ packet events (assuming no errors on the one-to-one communication) versus $6N + 2$ for a simple retransmission. As alluded to previously, note that the same cost is incurred regardless of whether all-reliable one-to-many ***Send*** is supported by the kernel or implemented by the sender on top of multiple-response one-to-many ***Send***, as provided by the kernel.

The above method uses standard positive acknowledgement and retransmission for reliable delivery. This guarantees that after a number of retransmissions the sender knows that the message is delivered to all operational processes in the group, and that the other members are currently inaccessible. An alternative strategy provides a somewhat different form of reliability: it guarantees that every message will eventually be

received by each process in the group (assuming the process expresses an interest in receiving the message). This scheme, analogous to publishing used in real world one-to-many communication, places a greater burden on the receivers. Basically, information to a group, the *subscribers*, is filtered through the *publisher* which collates and numbers the information before issuing it to the subscribers. Just as with a real world publication, it is the subscribers that implement the reliable reception (to the degree they require), not the publisher implementing reliable delivery. In particular, a subscriber notices that an *issue* has not been received by a gap in the issue numbers of issues received or because a new issue was not received in the expected time interval, in the case of regular publications. On detecting a missing issue, the subscriber may request the missing *back issue* from the publisher. Issue numbers are also used to detect receiving duplicate issues. Clearly, the publication model can be implemented using a combination of a reliable one-to-one ***Send*** to the publisher, unreliable one-to-many ***Send*** from the publisher to the subscribers, and reliable one-to-one ***Send*** from the subscribers to the publisher to obtain back issues.

A slight variant on the above uses a *logging process* instead of a publisher. Each group has a logging process. When a notification is sent to the group, by arrangement only the logging process responds. The message is retransmitted until the logging process responds thus providing reliable transport to the log. A group member that is concerned about a missed group message can consult the logging process to extract the message. This provides more immediate communication with the group (avoiding the delay of first sending to the publisher on each message) but makes it difficult to impose a strict ordering on messages received from different senders.

A further variant on this scheme by Chang and Maxemchuck [12] is described in Section 6.7.

6.4.4. Groups

The definition of group is *open* in a number of ways. First, processes from outside the group may send to the group using the group identifier. This is required to allow a client process to send to a group of servers, a common use in our experience. A group model whereby only group members are allowed to send to the group can be derived from the basic model by having member processes discard all messages but those believed to be originated within the group.

Second, any process can join or leave a group. Although this is fine as a basic model, without any additional mechanism it poses a serious protection problem. For instance, a process might join a group and disrupt its proper operation by sending bogus replies to group messages. We recognize that control is needed over group membership and plan to implement a means of refusing membership either based on user identification or objection from existing members of the group.

6.5. Implementation

We describe an implementation for the 3 Mb Ethernet and the 10 Mb Ethernet. The implementation deals with the communication aspect of multicast, and the allocation and management of multicast addresses and group identifiers.

6.5.1. Allocation of Group Identifiers and Multicast Addresses

When a ***AllocateGroupId()*** is executed, a unique group identifier is allocated and a multicast address is associated with it. The allocation of group identifiers and their mapping to multicast addresses is subject to the following two requirements:

1. The group identifier name space and the process identifier name space need to be strictly disjoint, with the confines of the name space offered by a 32-bit identifier.

2. The multicast address needs to be derivable from the group identifier. When a ***JoinGroup(groupId.pid)*** is executed, the kernel has to instruct the Ethernet interface to accept packets for the corresponding

multicast host address[29]. In other words, one cannot rely on inferring the mapping from incoming packets as for point-to-point communication (See Section 5.4.2), since the host address needs to be known before any messages addressed to the group can be received.

Additionally, the following properties are desirable:

1. The test whether an identifier is a process identifier or a group identifier should be simple and efficient.
2. As for process identifiers, we would like to allow for distributed generation of group identifiers while still guaranteeing their uniqueness.
3. Finally, we would like to achieve the situation where a multicast packet resulting from a group ***Send*** does not cause any processor utilization on processors which have no members in the group. This requires a one-to-one mapping from groups to multicast addresses.

In practice, group identifiers are allocated as follows. The most significant bit of a group identifier is always on, while the most significant bit of a process identifier is always off. This reduces the test whether a 32-bit identifier is a group identifier or a process identifier to a simple bit test. This multicast bit reduces the available name space for group identifiers to 31 bits, allowing for 2^{31} unique identifiers. Within a name space of that size, an acceptable degree of uniqueness can be achieved by simply letting each host pick a random number when it needs to allocate a group identifier. The probability of a collision is approximately m^2/n, where m is the number of groups and n is the size of the name space (See Section 5.3.2). For reasonable values of m, this probability is quite small. For instance, for m = 32, the probability of a collision is 0.0001%. If deemed necessary, this probability could be lowered even further using a collision detection technique similar to the one used for detecting collisions of logical host identifiers.

The allocation of multicast addresses is evidently dependent on the particular network used. In the 10 Mb Ethernet specification [27], large blocks of multicast addresses can be assigned to particular vendors. Thus, the ideal of having a separate multicast address for each group could be achieved[30]. Pending the assignment of such a block, we have taken the liberty of using a (static) set of multicast addresses. N bits of the group identifier indicate the corresponding multicast address, through a table lookup. On our 3 Mb Ethernet, we allocate a limited number of host addresses (8) that are currently unused, and proceed as with the 10 Mb[31].

6.5.2. Joining and Leaving a Group

When a ***JoinGroup(groupId,processId)*** is executed, the kernel of the machine on which the process with process identifier ***processId*** resides allocates a group descriptor, containing the mapping from group identifier to process identifier. This mapping only needs to be maintained on that particular machine. If the ***JoinGroup(groupId,processId)*** is executed on a machine different from the one on which the process with process identifier ***processId*** resides, the request is forwarded between the kernel on which the request was issued and ***processId's*** kernel by the interkernel protocol. Leaving the group results in the group descriptor being deleted.

6.5.3. Sending to a Group

When a ***Send*** is done to a group, the message packet is sent to the associated multicast address derived from the group identifier. The simple check for a group identifier is important in efficiently detecting and handling this case. In particular, it allows one-to-many communication to be implemented with absolutely minimal overhead for one-to-one communication (one extra bit test). If no reply is expected, the process is not

[29] By default, the Ethernet interface is only open for broadcast and for the local network address.

[30] Currently available interfaces severely limit the number of multicast addresses that can be received.

[31] Our 3 Mb network interface can be opened for any subset of the 256 possible 3 Mb network addresses.

blocked. Otherwise, the process is blocked as for a normal ***Send*** (with the possibility of retransmissions, etc.) until the first reply comes in, which is returned by the ***Send***.

In the case of multiple replies, the second and subsequent replies are buffered in the kernel and queued for reception by the sender using ***GetReply***. The next invocation of ***Send*** discards all queued replies[32]. For each process, we maintain a *sequence number*, which is incremented on every ***Send***. This allows us to discard delayed replies to earlier message transactions, and ensures that only replies to the current message transaction are queued in the kernel and eventually returned to the sender.

6.5.4. Receiving a Message from a Group Send

When the kernel receives a multicast packet, it checks the group descriptors to determine which local processes belong to this group. This check is done quickly by using a hash table. A copy of the message is linked into the message queue of each of the member processes (identical as for one-to-one messages). This mechanism also works for group members on the same machine as the originator of the message so no extra code is required for local members of a group[33].

6.5.5. Replies to a Group Send

If a group message has the no-reply bit set, replies to this message are short-circuited on the machine from where the ***Reply*** is executed and cause no network traffic. For one reply messages, replies are always transmitted but only the first one to arrive at the sender has effect. For multiple replies, the kernel attempts to receive and buffer all reply messages until the end of the associated message transaction, subject to available buffer space.

6.5.6. Performance

Ideally, we would like to achieve the situation where delivery of a message to N processes, using one-to-many ***Send*** is N times faster than using repeated one-to-one ***Sends***. Additionally, the cost should be comparable to using hardware multicast for the same purpose. In our current implementation, a 0-reply group ***Send*** takes 0.96 milliseconds while a 1-reply group ***Send*** takes basically the same time as a one-to-one ***Send***, namely 2.2 milliseconds. The time for a multiple-reply group ***Send*** should be the same to get the first reply and then roughly 0.2 millisecond per message to get the other replies assuming they have arrived when ***GetReply*** is called.

6.6. Applications

Conventional applications of broadcast and multicast can be built nicely on top of our group communication mechanism, thereby removing all network-related code from these applications, as discussed below. We also describe the use of one-to-many communication in highly parallel distributed computations.

6.6.1. Amaze: A Multi-Player, Multi-MachineGame

Amaze [8] is a multi-player game program that runs on a set of personal workstations connected by a local network. Each player maneuvers his representative "monster" through a maze with the objective of shooting the monsters controlled by other players without allowing his own monster to be shot. Each player has a personal workstation running a copy of Amaze, providing local display of the maze and monsters, and communication with other player workstations through a local network. Only the workstations of

---

[32] A timer with a suitably large timeout interval takes care of the case when no further ***Sends*** are done.

[33] This assumes that the network interface is capable of receiving its own packets. Not all interfaces have this desirable property.

participating players can be relied on to maintain the state of the game and players are free to join and quit at arbitrary times. Thus, there cannot be a single site for the global game state, which must therefore be replicated. This problem is quite typical of a number of distributed programming applications and therefore of general interest.

Our previous experience with distributed games stems from using Xerox Alto computers which run multi-player game programs such as Maze Wars. Maze Wars, like other games run on the Altos, uses Ethernet multicast to communicate game state updates to the workstations running the game. A state update is sent to a particular multicast address for the game. All players listen for updates on that multicast address. Amaze is an attempt to build a more network-independent multi-machine game.

The original implementation of Amaze relies entirely on the one-to-one communication primitives of the V kernel. A *game manager* process maintains the local copy of the game state on each workstation and also ensures that the workstation display accurately reflects the current game state. The game manager performs state updates in response to messages from its *helper* processes: a timer process, a keyboard reader and one *status inquirer* process per remote player. A remote state update is reported to the game manager by a status inquirer that is dedicated to reading and reporting the state of a particular monster running on another machine. The inquirer sends a message to the remote monster's manager requesting a status update and then pauses to await the event. The remote game manager only replies when a change of state occurs locally that has not previously been reported[34]. Having obtained the update, the inquirer sends a message reporting it to its own manager.

Using the group mechanism, no status inquirers are necessary at all. The game manager simply joins the group and listens to updates from other players. It sends state updates effected locally to the group, using no-reply ***Sends***. Additionally each game manager periodically sends its state to the group (regardless of whether it has a state update to report) to indicate that its workstation is still participating in the game.

The implementation using the group mechanism compares favorably with the original point-to-point implementation on several counts:

1. In terms of cost, the group implementation is better both in network bandwidth as well as in the overall number of packet events.

   For a game with n participants, a state update requires 1 network packet, compared to 2(n-1) using point-to-point connections. The number of packet events is n+1 per state update, compared to 4(n-1) before. The "steady-state" traffic (when there are no state updates) accounts for n packets per timeout interval when using the group mechanism compared to 2n(n-1) originally. The number of packet events due to steady-state traffic is n^2 as opposed to 4n(n-1).

   On the down side, the application must itself ensure periodic transmission during quiet time intervals. For one-to-one communication, the kernel takes care of such chores.

2. Fewer processes are necessary: 3 when using the group mechanism (timer, keyboard and game manager) compared to n+2 when using point-to-point communication (n-1 additional status inquirers).

3. The interval between the periodic state updates can be freely set by the application and is not bound by the retransmission interval of the kernel. The fact that the latter interval is fairly long (2.5 sec) was a source of annoying temporary inconsistencies in the original implementation.

Note that while using multicast, the new Amaze program does not contain any network-specific code (as the Alto games do) and therefore remains relatively portable to other network technologies. In fact, the game has been ported from the 3 Mb Ethernet to the 10 Mb Ethernet without any modification.

---

[34] Although no process-level message traffic is present during these "quiet" periods, the sender's kernel periodically polls the replier's kernel in order to distinguish a "quiet" state from the case where the replier is dead. This causes 2 network packets to be transmitted during each polling interval (2.5 seconds).

The game program is an example of an application of one-to-many communication that falls under the general category of *unreliable notification.* It is not important that all notifications arrive at all destinations, since subsequent updates subsume previous ones (Each update carries absolute state information.) A number of practical systems exhibit this property. However, some applications require that all notifications reach all accessible destinations, such as for distributed database updates. These applications would use the reliable notification techniques discussed earlier.

6.6.2. Locating Servers

The previous example fits under the notification paradigm. The other major use of one-to-many communication is for queries.

Locating a service in the V-System involves finding out the process identifier of a process implementing the service. All processes implementing a particular service form a group representing that service. This group is identified by one of the predefined group identifiers (See Section 6.3). When a server starts up, it executes a ***JoinGroup(serverGroupId, myPid).*** Clients trying to locate a service then execute a ***Send(msg, serverGroupId),*** the requestcode of the message being ***Query***, one of the standard system requestcodes. By convention, all servers know how to process such a ***Query*** request and respond to it appropriately.

The original V kernel had special kernel operations for the same purpose, ***SetPid*** and ***GetPid.*** For a definition of these primitives, see Section 3.2; for a description of their implementation, see Section 5.6.3. A number of problems were identified with this approach. First, special kernel support was present although it did not seem inherently necessary. Instead, it was merely a result of the absence of one-to-many communication as a generally available facility at the kernel interface. These extra primitives did not only require extra code and data structures in the kernel, but also special packet types in the interkernel protocol, and associated packet handlers. Second, the approach was prone to annoying inconsistencies. Once a process was registered, it remained registered until its registration was overridden by another process. So, the kernel would occasionally return an invalid process identifier. The situation was somewhat ameliorated by having the kernel check the validity of the process identifiers it returned. However, this precluded registering remote processes or groups (because the kernel itself cannot check validity of remote processes). The approach was also found to be unnecessarily inflexible. Only a scope modifier could be added to either a server registration or a client location request. Besides, it was an all-or-nothing approach: a server was either registered or not, it could not selectively respond or refuse to respond to certain requests. Finally, the approach was somewhat vulnerable to bogus processes registering themselves.

Using a one-to-many query to the servers themselves has many advantages over the original mechanism. First, no kernel support is necessary beyond what is already available. Second, there is no danger of permanent inconsistency since the servers themselves respond to queries. Third, the mechanism allows the servers great flexibility in responding to lookups. For instance, a server can respond only to authorized users, when its load is not too high, etc.

With respect to one-to-many communication, it allows us to demonstrate an example of both one-reply and multiple-reply query. Simple applications are typically interested in locating a generic service, without worrying much about which particular instance of that service they obtain as the result of their query. The one-reply option of query is adequate for their purposes. More sophisticated applications might elect to receive a number of replies. The application involving remote execution discussed in Section 6.4 is a good example. Additionally, the multiple-reply mode of operation alleviates somewhat the problem of a bogus process registering itself as a certain kind of server. Such a process can still join a group and respond to ***Query*** requests, but since many replies can be received to such a ***Query***, the client at least gets a choice of server.

6.6.3. Distributed Computation

A class of applications of special interest to us involves large-scale, highly parallel distributed computations. Here we envision many processes running on separate processors (possibly provided by separate workstations) working in parallel to solve a CPU-intensive problem. One-to-many communication is used to query the

progress of other processes as well as notify other processes of new insights discovered. In general, we structure the sharing of the growing program knowledge base in a fashion similar to the way in which researchers share their knowledge. Results and queries may go one-to-one to closely cooperating colleagues, as with "private communications". Important results are submitted for *publication* to a publisher process. The publisher then decides whether or not to distribute that information. More importantly, the publisher takes on the job of distribution and maintenance of back issues, freeing the researcher to continue its work.

As an example of a distributed computation using one-to-many communication, consider a parallel α-β search (See Figure 6-2). Several searcher processes explore different branches of the game tree with some initial search window. All these processes belong to a single group, together with a manager process. As the individual processes evaluate their subtrees, they obtain narrower bounds on the overall search window. These narrower bounds are of interest to all searchers since they might prune parts of other subtrees as well. So any new bounds are announced to the group by one-reliable notification. By arrangement, only the publisher ever responds to a message sent to the group, thereby providing one-reliable notification to the publisher process.

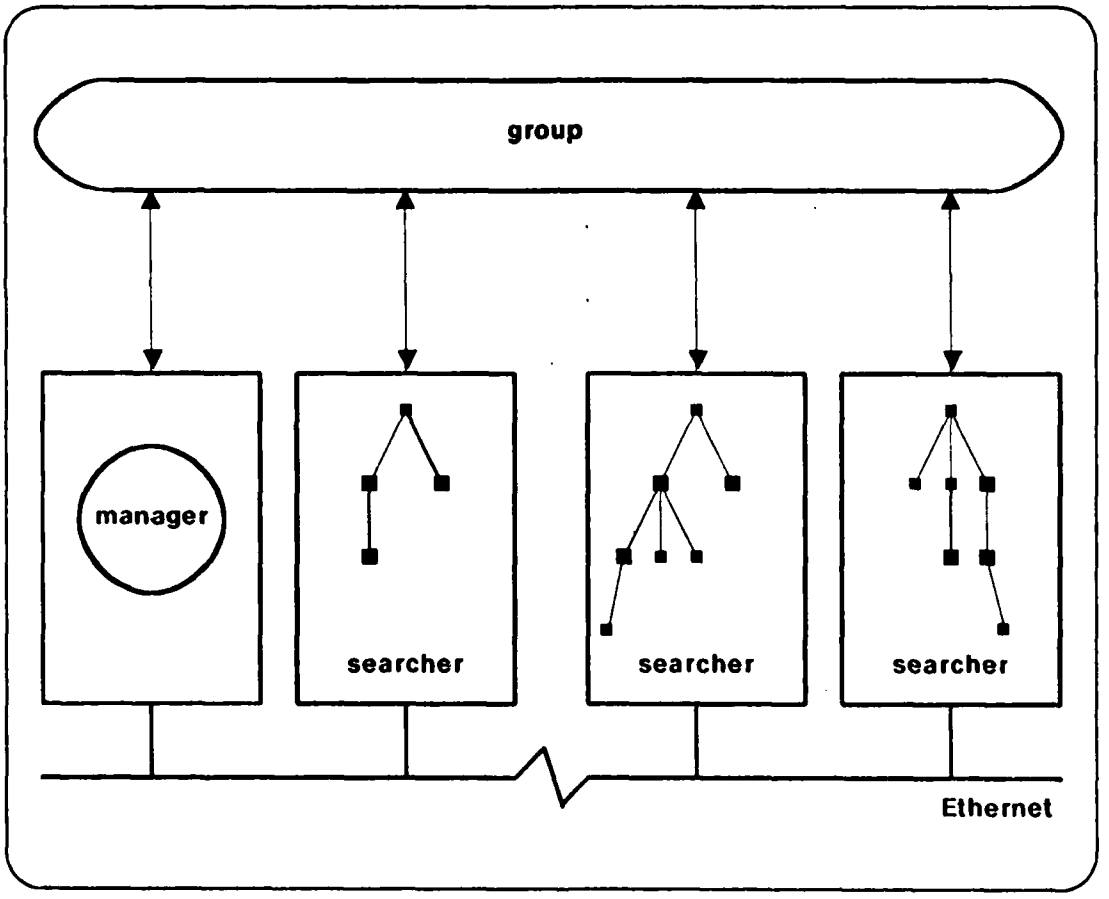


Figure 6-2: α-β Search on a Collection of Workstations

Note that the one-reliable aspect of the notification is essential in order to ensure that the final result of the computation is correct: it makes sure that the publisher sees all results, and deduces correctly the best move for the given position. If one of the searcher processes misses a message, this does not affect the correctness of

the algorithm. It might cause this process to do some extra work but the final result will remain the same. Similarly, a program to solve the traveling salesman problem could use simple notification to communicate the current least cost path to all parallel searchers, reducing the exploration of inferior routes.

6.7. Related Work

Since the subject of one-to-many communication is somewhat distinct from the subject matter of the rest of this thesis, we include in this chapter a separate section describing related work on broadcast and multicast communication.

One area of related work concerns the provision of broadcast and multicast at the data link and network levels. In particular, the Ethernet [54] broadcast network requires only multicast addressing and filtering in the hosts to provide multicast delivery. Implementing broadcast and multicast on point-to-point networks or internetworks has been covered by a number of authors, including Dalal and Metcalfe [25], Wall [76] and Boggs [10] All this work addresses the efficient multicast delivery of packets to hosts, and thereby provides the necessary underlying mechanism for efficient one-to-many *interprocess* delivery as addressed by the V-System.

The use of the broadcast and multicast capabilities of local networks has been the subject of other related work, Boggs' thesis [10] being the primary reference. Our work focuses on proposing an application-level interface to one-to-many communication, a subject which to the best of our knowledge has not been addressed in previous work.

Finally, Chang and Maxemchuck [12] describe a reliable broadcast protocol similar to our publication model. They define reliable broadcast to a set of machines to guarantee that all machines receive the same sequence of notification messages. Summarizing their implementation, each notification is broadcast to all machines A *token host* assigns a sequence number to the message (incremented by one for every message), and then broadcasts an acknowledgement containing this sequence number. Again, as in the publication model, the onus is on the receivers to request retransmissions from the token machine. The broadcast acknowledgement seems difficult to formalize in the request-response paradigm of the V world. Additionally, their work contains algorithms for transfer and generation of tokens, and correctness proofs of the protocols involved.

6.8. Chapter Summary

One-to-many communication is useful in distributed systems in the absence of global shared memory, which provides this facility implicitly. The performance of such a facility is important for highly parallel distributed computation. With this motivation, we have extended the V kernel to provide a simple group ***Send*** form of one-to-many communication with the option of zero, one or multiple replies. This simple mechanism supports the common forms of query and notification and can be used by the application to implement fully reliable notification and query if required. We argued against implementing fully reliable multicast directly in the kernel. Finally, we have described some initial applications.

Several questions remain unanswered at this point. First, as we already pointed out, a protection issue is raised by the very open structure of process groups. We plan to implement a mechanism whereby membership of a group can be refused based on lack of user authentication or based on objections by other group members. Second, we have ignored the interference of one-to-many communication with segment access as used for ***MoveTo*** and ***MoveFrom***. Currently, we allow concurrent read access to a segment. If write access is given to a segment, the first replier is given exclusive access to the segment. We do not have much evidence to indicate whether this semantics is appropriate. Finally, we are interested in exploring the potential for large-scale parallel computation of a system consisting of processors interconnected by a broadcast network.

—7—
Conclusions

7.1. Summary

In this thesis we have presented the design, the implementation and the evaluation of a transparent message-based interprocess communication mechanism on a local area network.

Chapter 3 contains the definition of the interprocess communication primitives and measurements of their performance on SUN workstations interconnected by a 3 or a 10 Mb Ethernet network. We introduced the notion of *network penalty* as the minimum cost for transferring data between machines over a network, given a particular combination of processor, network and network interface. Based on this notion, we estimated lower bounds on the elapsed time for the intercommunication primitives when implemented between machines in the given configuration. We showed that the elapsed time for a message exchange is only one millisecond higher than the lower bound, and that the elapsed time for transferring one kilobyte of data is less than one millisecond above the lower bound. Furthermore, we have compared the performance of file access and pipes when implemented on top of the message passing primitives to estimates of the performance of those applications when supported by a dedicated protocol.

The performance of network file access was further investigated in Chapter 4. The measurements in Chapter 3 were done in an otherwise idle environment and in a particular hardware configuration. To assess performance under load, and to investigate how performance would change between different configurations, we built a queueing network model of file access from diskless workstations over a local network to a set of file servers. Results from the model indicate that transparent file access is possible without significant performance degradation, for moderate numbers of workstations. Under a plausible set of assumptions (including those present in the baseline configuration), the file server CPU is shown to be the bottleneck. We have explored a number of options whereby this bottleneck could be relieved. In particular, using large client-server interaction sizes, introducing a file server cache and using a second file server offered good potential for improving performance under high load.

The protocol underlying the message passing primitives was studied in Chapter 5. Efficiency was a primary concern during the design of the protocol and its implementation. This emphasis is reflected in this chapter in discussing methods for process identifier generation, process location, message passing and data transfer. Of particular interest was the discussion of the efficiency of moving large amounts of data reliably across a local area network. For the low error rates typical for local area networks, it was shown that the expected time of such a transfer is almost identical to the elapsed time when no errors occur, regardless of the retransmission strategy. While the retransmission strategy has little effect on the expected time for the transfer under the assumption of low error rates, it has a dominant effect on the standard deviation of the elapsed time.

Finally, in Chapter 6, we proposed a mechanism for integrating one-to-many communication into message-passing, drawing partially on the broadcast and multicast capabilities of local networks. We have discussed the issues involved in addressing groups of processes and in providing (different degrees of) reliability for one-to-many communication. In particular, we have argued against implementing fully reliable one-to-many communication at the kernel level. Some initial applications of one-to-many communication were explored as well.

7.2. Future Research

Several issues were left unanswered by this thesis. An important class of remaining open questions is concerned with the validity of the results in different environments. We briefly survey some of the issues.

7.2.1. Internetworking

The current implementation of the V-System runs on a single local area network. No provisions are made in the current implementation for operating in an internet environment. Encapsulating the V interkernel packets in internet packets rather than Ethernet data link layer packets provides only the first step in this direction. As indicated in Chapter 5, the process identifier allocation mechanism, and to a lesser degree, the process location mechanism, rely on the capability of being able to broadcast packets under certain circumstances. Most current internet protocols do not admit broadcasting, so either extensions to the internet protocols must be explored or the V interkernel protocol must be adapted. Second, both the latency and the error rate on an internet are higher than on a local area network. The robustness of the protocol in the face of higher error rates and longer delays must be improved. Whether all of this can be done without unduly affecting performance (on a local network) remains an open question.

7.2.2. Network Demand Paging

In Chapter 4 we have discussed in great length the performance of sequential file access across a local network. Network demand paging poses a different set of challenges, in terms of workload as well as in terms of the benefits of buffering, etc. Also, while the advantages of a shared file server are quite clear, the case for a shared paging server seems less strong, since paging, unlike file access, has little notion of sharing. The sharing aspects of a paging server thus seem to get in the way of performance without providing much benefit in return.

During the design of the protocol, several performance-related decisions were made based on intuitive notions about (avoiding) buffering. The cost of buffering changes drastically when demand-paged machines are considered, where the copy operation, inherent in buffering, can potentially be subsumed by mapping and unmapping pages. At the time of writing, the system has been ported to a 68010-based machine, capable of demand paging, although the current implementation does not take advantage of this capability.

7.2.3. Smart Network Interfaces

Finally, more than once we lamented the inadequacies of currently available network hardware. Suggested modifications ranged from double buffering to dedicated network interface processors. Double buffering would potentially be able to provide data rates comparable to the network data rate. Intelligent scatter-gather DMA interfaces are able to reduce processor utilization on transmission, but they seem to be of little advantage on reception of packets. Programmable network interfaces have the potential for further reducing the processor utilization for network interprocess communication. Careful hardware-software integration is required to be able to take maximum advantage of these sophisticated interfaces.

References

1. 3-Com Corporation. *3C400 Multibus Ethernet Controller Reference Manual.* 1982.

2. J.E. Allchin and M.S. McKendry. Synchronization and Recovery of Actions. Proceedings of the Second Conference on Principles of Distributed Computing, ACM, August, 1983, pp. 31-44.

3. G.T. Almes, A.P. Black, E.D. Lazowska and J.D. Noe. The Eden System: A Technical Review. Tech. Rept. 83-10-05, University of Washington, October, 1983.

4. G.T. Almes and E.D. Lazowska. The Behavior of Ethernet-like Computer Communication Networks. Proceedings of the Seventh Symposium on Operating Systems Principles, ACM, December, 1979, pp. 66-81.

5. F. Baskett, J.H. Howard, and J.T. Montague. Task Communication in DEMOS. Proceedings of the Sixth Symposium on Operating Systems Principles, ACM, November, 1977, pp. 23-31.

6. A. Bechtolsheim, F. Baskett, and V. Pratt. The SUN Workstation Architecture. Tech. Rept. 229, Computer Systems Laboratory, Stanford University, March, 1982.

7. E.J. Berglund, P. Bothner, K.P.Brooks, D.R. Cheriton, D.R. Kaelbling, K.A. Lantz, T.P. Mann, R.J. Nagler, W.I. Nowicki, M.M. Theimer and W. Zwaenepoel. V-System Reference Manual. Computer Systems Laboratory, Stanford University, 1983.

8. E.J. Berglund and D.R. Cheriton. Amaze: A Distributed Multi-Player Game Program using the Distributed V Kernel. Proceedings of the Fourth International Conference on Distributed Systems, IEEE, May, 1984, pp. 248-255.

9. A.D. Birrell and B.J. Nelson. "Implementing Remote Procedure Calls." *ACM Transactions on Computer Systems 2*, 1 (February 1984), 39-59.

10. D.R. Boggs. *Internet Broadcasting.* Ph.D. Th., Stanford University, October 1983. Also Technical Report CSL-83-3, Xerox PARC

11. D.R. Boggs, J.F. Shoch, E.A. Taft, and R.M. Metcalfe. "Pup: An Internetwork Architecture." *IEEE Transactions on Communications COM-28*, 4 (April 1980), 612-624.

12. J.M. Chang and N.F. Maxemchuck. Reliable Broadcast Protocols. Computer Technology Research Laboratory, AT&T Bell Laboratories, January, 1983. To appear in ACM Transactions on Computer Systems

13. D.R. Cheriton. Distributed I/O using an Object-based Protocol. Tech. Rept. 81-1, Computer Science Department, University of British Columbia, January, 1981.

14. D.R. Cheriton. *The Thoth System: Multi-Process Structuring and Portability.* North-Holland/Elsevier, 1982.

15. D.R. Cheriton. "The V Kernel: A Software Base for Distributed Systems." *IEEE Software 1*, 2 (April 1984), 19-42.

16. D.R. Cheriton, M.A. Malcolm, L.S. Melen, and G.R. Sager. "Thoth, a Portable Real-time Operating System." *Communications of the ACM 22*, 2 (February 1979), 105-115.

17. D.R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. Proceedings of the Ninth Symposium on Operating System Principles, ACM, October, 1983, pp. 129-140.

18. D.R. Cheriton and W. Zwaenepoel. One-to-many Communication in the V-System. Abstract appears in Proceedings SigComm '84. To appear in ACM Transactions on Computer Systems

19. D.D. Clark. System Requirements for Distribution. Proceedings of the Workshop on Fundamental Issues in Distributed Computing, December, 1980, pp. 250-255.

20. D.D. Clark. Modularity and Efficiency in Protocol Implementation. RFC 817, NIC, July, 1982.

21. D.D. Clark and L. Svobodova. Design of Distributed Systems Supporting Local Autonomy. Proceedings of Spring CompCon Conference, IEEE, February, 1980, pp. 438-444.

22. J.H. Clark and T. Davis. "Workstation Unites Real-time Graphics with Unix, Ethernet." *Electronics* (October 1983), 113-119.

23. R.P. Cook. "StarMod -- A Language for Distributed Programming." *IEEE Transactions on Software Engineering SE-6*, 6 (November 1980), 563-571.

24. O.J. Dahl, B. Myhrhaung and K. Nygaard. *The Simula 67 Common Base Language.* Norwegian Computer Center, Oslo, Norway, 1968.

25. Y.K. Dalal and R.M. Metcalfe. "Reverse Path Forwarding of Broadcast Packets." *Communications of the ACM 21*, 12 (December 1978), .

26. R.C. Daley and J.B. Dennis. "Virtual Memory, Processes, and Sharing in Multics." *Communications of the ACM 11*, 5 (May 1968), 306-312.

27. Digital Equipment Corporation, Intel Corporation and Xerox Corporation. *The Ethernet: A Local Area Network - Data Link Layer and Physical Layer Specifications, Version 1.0.* 1980.

28. R. A. Finkel. Private Communication. 1984

29. A. Goldberg, G. Popek and S. Lavenberg. A Validated Distributed System Model. Performance '83, IFIP WG 7.3, September, 1983, pp. 251-268.

30. T.A. Gonzalves. Private Communication. 1983

31. E. Harslem and L.E. Nelson. A Retrospective on the Development of Star. Proceedings of the Sixth International Conference on Software Engineering, IEEE, September, 1982.

32. P. Heidelberger and K.S. Trivedi. "Queueing Network Models for Parallel Processing with Asynchronous Tasks." *IEEE Transactions on Computers C-31*, 11 (November 1982), 1099-1109.

33. E. Holler. The National Software Works (NSW). In *Distributed Systems: Architecture and Implementation*, B.W. Lampson, M. Paul and H.J. Siegert, Ed.,Springer-Verlag, 1981, pp. 421-445.

34. B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, 1978.

35. B.W. Lampson and K.A. Pier. A Processor for a High-Performance Personal Computer. Proceedings of the Seventh Symposium on Computer Architecture, ACM/IEEE, May, 1980, pp. 146-160.

36. K.A. Lantz, K.D. Gradischnig, J.A. Feldman, and R.F. Rashid. "Rochester's Intelligent Gateway." *Computer 15*, 10 (October 1982), 54-68.

37. K.A. Lantz and W.I. Nowicki. Virtual Terminal Services in Workstation-based Distributed Systems. Proceedings of the 17th Hawai Conference on System Sciences, ACM/IEEE, January, 1984, pp. 196-205.

38. K.A. Lantz and W.I. Nowicki. "Structured Graphics for Distributed Systems." *ACM Transactions on Graphics 3*, 1 (January 1984), 23-51.

39. K.A. Lantz, W.I. Nowicki and M.M. Theimer. Network Factors Affecting the Performance of Distributed Applications. Proceedings SigComm '84, ACM, June, 1984, pp. 116-123.

40. E.D. Lazowska. Private Communication. 1984

41. E. Lazowska, H. Levy, G. Almes, M. Fischer, R. Fowler and S. Vestal. The Architecture of the Eden System. Proceedings of the Eighth Symposium on Operating System Principles, December, 1981, pp. 127-136.

42. E.D. Lazowska, J. Zahorjan, D.R. Cheriton and W. Zwaenepoel. File Access Performance of Diskless Workstations. Department of Computer Science, University of Washington, June, 1984.

43. E. Lazowska, J. Zahorjan, S. Graham and K. Sevcik. *Quantitative System Performance: Computer System Analysis using Queueing Network Models.* Prentice-Hall, 1984.

44. P.J. Leach, P.H. Levine, B.P. Douros, J.A. Hamilton, D.L. Nelson and B.L. Stumpf. "The Architecture of an Integrated Local Network." *IEEE Journal on Selected Areas in Communications SAC-1*, 5 (November 1983), 842-857.

45. T.J. Leblanc. *The Design and Performance of High-level Language Primitives for Distributed Computing.* Ph.D. Th., University of Wisconsin at Madison, December 1982.

46. T.J. Leblanc and R.P. Cook. An Analysis of Language Models for High-Performance Communication in a Local Network. Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, ACM, June, 1983, pp. 65-72.

47. B. Liskov. Primitives for Distributed Computing. Proceedings of the Seventh Symposium on Operating System Principles, December, 1979, pp. 33-42.

48. B.H. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust Distributed Programs. Proceedings of the Ninth Symposium on Principles of Programming Languages, ACM, January, 1982, pp. 7-19.

49. T.W. Lockhart. The Design of a Verifiable Operating System Kernel. Tech. Rept. 79-15, Department of Computer Science, University of British Columbia, November, 1979.

50. G.W.R. Luderer, H. Che, J.P. Haggerty, P.A. Kirslis, W.T. Marshall. A Distributed UNIX System based on a Virtual Circuit Switch. Proceedings of the Eighth Symposium on Operating System Principles, December, 1981, pp. 160-168.

51. M.A. Malcolm. Private communication. 1983

52. M.K. McKusick, W.N. Joy, S.J. Leffler and R.S. Fabry. A Fast File System for Unix. Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, July, 1983.

53. J.M. McQuillan and D.C. Walden. "The Arpanet Design Decisions." *Computer Networks 1*, 5 (September 1977).

54. R.M. Metcalfe and D.R. Boggs. "Ethernet: Distributed Packet Switching for Local Computer Networks." *Communications of the ACM 19*, 7 (July 1976), 395-404. Also Technical Report CSL-75-7, Xerox Palo Alto Research Center, reprinted as CSL-80-2.

55. Paul V. Mockapetris. Analysis of Reliable Multicast Algorithms for Local Networks. Proceedings of the Eighth Data Communications Symposium, ACM, Ocotober, 1983, pp. 150-157.

56. R.M. Needham and A.J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley, 1982.

57. B.J. Nelson. *Remote Procedure Call.* Ph.D. Th., Carnegie-Mellon University, 1981. Also Technical Report CSL-81-9, Xerox Palo Alto Research Center.

58. E.I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, 1972.

59. G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, G. Thiel. LOCUS: A Network Transparent High Reliability Distributed System. Proceedings of the Eighth Symposium on Operating System Principles, December, 1981, pp. 169-177.

60. J.B. Postel. "Internetwork Protocol Approaches." *IEEE Transactions on Communications COM-24*, 4 (April 1980), 604-611.

61. M.L. Powell, B.P. Miller and D.L. Presotto. DEMOS/MP: A Distributed Operating System. Technical Report in preparation, University of California at Berkeley

62. B. Randell. "The Newcastle Connection: Unix United." *Software: Practice and Experience 12*, 6 (June 1982), 304-315.

63. R. Rashid and G. Robertson. Accent: A Communication Oriented Network Operating System Kernel. Proceedings of the Eighth Symposium on Operating System Principles, December, 1981, pp. 64-75.

64. M. Reiser and S.S. Lavenberg. "Mean Value Analysis of Closed Multichain Queueing Networks." *Journal of the ACM 27* (April 1980), 313-322.

65. D.M. Ritchie and K. Thompson. "The UNIX timesharing system." *Bell System Technical Journal 57*, 6 (July/August 1978), 1905-1929.

66. L.A. Rowe and K.P. Birman. "A Local Network based on the Unix Operating System." *IEEE Transactions on Software Engineering SE-8*, 2 (March 1982), 137-146.

67. P. Schweitzer. Approximate Analysis of Multiclass Closed Networks of Queues. International Conference of Stochastic Control and Optimization, 1979.

68. W.D. Sincoskie and D.J. Farber. "SODS/OS: A Distributed Operating System for the IBM Series/1." *SIGOPS Operating Systems Review 14*, 3 (July 1980), 46-54.

69. W.D. Sincoskie and D.J. Farber. The Series/1 Distributed Operating System: Description and Comments. Proceedings of the Fall COMPCON Conference, IEEE, September, 1980, pp. 579-584.

70. M.H. Solomon, R.A. Finkel. The Roscoe Distributed Operating System. Proceedings of the Seventh Symposium on Operating System Principles, December, 1979, pp. 108-114.

71. A.Z. Spector. *Multiprocessing Architectures for Local Networks.* Ph.D. Th., Stanford University, 1981. Also Technical Report STAN-CS-81-874, Department of Computer Science, Stanford University

72. A.Z. Spector. "Performing Remote Operations Efficiently on a Local Computer Network." *Communications of the ACM 25*, 4 (April 1982), 260-273.

73. C.P. Thacker, E.M. McCreight, B.W. Lampson, R.F. Sproull and D.R. Boggs. Alto: A personal computer. In *Computer Structures: Principles and Examples*, D.P. Siewiorek, C.G. Bell and A. Newell, Ed.,McGraw-Hill, 1982, pp. 549-572.

74. R.H. Thomas. A Resource Sharing Executive for the Arpanet. Proceedings of the National Computer Conference, AFIPS, June, 1973, pp. 155-163.

75. B. Walker, G. Popek, R. English, C. Kline and G. Thiel. The LOCUS Distributed Operating System. Proceedings of the Ninth Symposium on Operating System Principles, ACM, October, 1983, pp. 49-70.

76. D.W. Wall. *Mechanisms for Broadcast and Selective Broadcast.* Ph.D. Th., Stanford University, June 1980.

77. J.E. White. A High-level Framework for Network-based Resource Sharing. Proceedings of the National Computer Conference, AFIPS, June, 1976, pp. 561-570.

78. M.V. Wilkes and R.M. Needham. "The Cambridge Model Distributed System." *SIGOPS Operating Systems Review 14*, 1 (January 1980), 21-29.

79. M.V. Wilkes and D.J. Wheeler. The Cambridge Digital Communications Ring. Proceedings Local Area Communication Network Symposium, NBS, May, 1979.

80. J. Zahorjan and E.D. Lazowska. Incorporating Load-Dependent Service Centers in MVA Models. Proceedings of the 1984 Sigmetrics Conference on Modeling and Measurement of Computer Systems, ACM, August, 1984.

81. W. Zwaenepoel and K.A. Lantz. "Perseus: Retrospective on a Portable Operating System." *Software: Practice and Experience 14*, 1 (January 1984), 34-48.

END

FILMED

6-86

DTIC